Experiments in distributed memory time warp

Simmonds, Robert W. J.

Award date:
1999

Awarding institution:
University of Bath

Experiments in Distributed Memory Time Warp

submitted by

Robert W. J. Simmonds

for the degree of PhD

of the

University of Bath

1999

ProQuest LLC
789 East Eisenhower Parkway
P.O. Box 1346
Ann Arbor, MI 48106-1346

Summary

Parallel Discrete Event Simulation (PDES) algorithms have been available for over a decade, yet they are still to gain acceptance from the general simulation community. Two problems have led to this reluctance to use PDES. Simulation models have been difficult to encode using the PDES systems available up to this time and it has proved difficult to tune the PDES simulators to work well with new simulation models. Due to this, a high level of expertise is required to program and tune new simulation models.

This thesis considers several issues that could make using a PDES system easier. It builds upon the Time Warp algorithm since I believe this provides the best basis for a general purpose PDES system. The main issues covered are hiding state saving from the modeller and providing an adaptive event scheduling scheme that allows new simulations to be run efficiently without the underling simulation kernel requiring lengthy tuning.

A distributed memory Time Warp system was written to support this work and the components of this system are explained. Two major components of the system are the transparent state saving scheme and the GVT management subsystem, both of which are explained at length. The factors that affect the behaviour of a Time Warp system are presented along with an explanation of why inefficient modes of behaviour can occur using Time Warp with some simulation models. This explains the need for event scheduling and throttling algorithms and existing algorithms that perform these tasks are explained. Then a new scheduling algorithm is presented along with details of the test implementation. Experimental results are presented comparing the new algorithm with the most often used existing algorithm.

Acknowledgements

First I would like to thank Liz Davies and John Fitch for their guidance and encouragement that led me to enter the postgraduate program. I also thank John as my supervisor, for his invaluable intellectual input and his help negotiating the rules and regulations associated with gaining a higher degree.

Next I would like to thank the members of the Denton group in the School of Mathematical Sciences at the University of Bath for allowing me to use their research equipment, to participate in their research discussions and for funding me to attend conferences. I therefore particularly thank the main academic participants Julian Padget and Russell Bradford and also my fellow researcher Andreas Kind.

I would like to thank Brian Unger and the TeleSim group at the University of Calgary. Brian allowed me time to finish my thesis while working in Calgary and gave me access to computing equipment that was not accessible to me at Bath. I also thank the MACI group at the University of Calgary and University of Alberta for allowing the use of their parallel computing resources.

I thank my fellow students and the staff within the School of Mathematical Sciences at the University of Bath for the free exchange of ideas that assisted me in this work. Finally I would like to thank my family who have supported me throughout this endeavour.

Contents

Chapter 1

Introduction

Computer simulation is a valuable tool for understanding complicated physical systems. The physical system in question could be a manufacturing production line, one or more military units, a large area computer network, or any number of other systems. These physical systems can be described in terms of system variables which evolve over time. The simulation of the system is simply a realization of a model that approximates the system and allows the system's behaviour to be understood for a given set of initial conditions, Seila [52].

Computer simulation can be split into two broad categories, *continuous time* and *discrete event*. In both types of simulation, the progression of time in the physical system is modelled by the progression of *virtual time* in the simulation.

A discrete event simulation models a system that only changes at distinct points in time. The changes in the physical system being modelled are represented by events in the simulation. Each event is represented by a record that contains a field holding the time that the event would occur in the physical system. Continuous time simulations treat time as a continuous variable and express system changes in terms of a set of differential equations involving the system state variables. In practice, the variables in continuous time simulations are evaluated at discrete, usually equally spaced points in virtual time. This update can be scheduled by an event, allowing systems modelled using continuous time to be integrated into a discrete event simulation.

A simulation executes until some pre-defined stopping condition has been met. The stopping condition could be a value of virtual time being reached, or some solution achieved. In a discrete event simulation the stopping condition could be met when there are no more events to execute.

1.1 Parallel Discrete Event Simulation

Parallel discrete event simulation (PDES) refers to running a single discrete event simulation on a parallel computer. The aim of running a simulation in parallel is to complete the simulation in a shorter time than is possible sequentially. This is only possible if events can be executed concurrently.

In this thesis, the simulation model is described in terms of the *logical process modelling methodology*, Chandy and Misra [8]. The system being modelled, described as the *physical system*, is partitioned into disjoint components called *physical processes* (PPs) that only interact by exchanging messages. A computer model is constructed by mapping the state of the physical processes to *logical processes* (LPs) in the simulation system. The messages in the physical model are mapped to timestamped *event messages* in the computer model. Each event message generated will cause an event to be scheduled at the virtual time represented by the message's timestamp, at the LP to which the message is sent. Note that a message could be sent from an LP to itself.

In a central event list sequential simulator, all events are executed in increasing timestamp order guaranteeing that causal relationships in the model are maintained. In a parallel simulator the events are distributed among the executives, with one executive allocated to each processor. To achieve speedup while still calculating the correct simulation results, a method is required to determine which events can be executed concurrently without breaking the causal relationships in the model.

A sufficient condition for the causal relationships to be maintained in the system as a whole, is that events are executed by each LP in monotonically non-decreasing timestamp order. This requirement is described as the *local causality constraint*, Fujimoto [22]. If a simulation algorithm does not maintain the local causality constraint, the effect is that of actions occurring later in the physical system effecting actions that occur earlier. This condition is known as a *causality error*. As long as a PDES algorithm maintains the local causality constraint, the simulation using this algorithm will give causally correct results.

1.1.1 PDES Algorithms

Parallel discrete event simulation algorithms are generally divided into two categories, *conservative* and *optimistic*. One further category of algorithm exists which are known as *Aggressive No Risk* (ANR) algorithms. An ANR algorithm combines ideas from both conservative and optimistic approaches. The first PDES algorithms used the conservative approach. Later optimistic algorithms were developed, with ANR algorithms being the latest category to appear. These categories can be best explained using the terms *risk* and *aggressiveness* as defined by Reynolds [49].

Definition 1 *An algorithm is* **aggressive** *if event messages are executed without knowing if they obey the local causality constraint.*

Definition 2 *An algorithm has* **risk** *if event messages are dispatched before the correctness of the event that caused them to be generated has been established.*

An event is known to be correct once it can be proved that it has been executed in accordance with the local causality constraint and, if such an event exists, the event whose execution caused this event to be generated has been proved to be correct. This is a recursive definition.

If an algorithm is aggressive, it may execute events in the incorrect order which could lead to a causality error. Therefore, aggressive algorithms must be able to detect when the local causality constraint has been violated and be able to recover when this situation is detected. If an algorithm uses risk, it is possible for event messages to be sent that do not relate to any action in the system being modelled. Therefore, algorithms using risk must be able to cancel events that are found to be incorrect and undo any changes that they have caused.

Conservative algorithms are not aggressive and don't use risk, while optimistic algorithms are aggressive and use risk. As the name implies, ANR algorithms are aggressive but do not use risk. This means that ANR algorithms have to be able to rollback, but never have to cancel event messages.

1.1.2 Conservative Algorithms

The first PDES algorithm was the conservative channel based Chandy-Misra-Bryant (CMB) algorithm developed independently by Chandy and Misra [8] and Bryant [7]. In this, event messages are passed along statically defined channels linking LPs. Event messages can only be passed along the channels in non-decreasing timestamp order. When a message arrives, the channel clock for the channel that the event arrives on is set to the value of the message's timestamp. Each LP can determine which events are safe to execute, i.e., which events can be executed without risking a causality error. This is done by finding the minimum of the clocks of all the input channels to the LP.

Just using the timestamps of event messages to advance channel times would lead to deadlock in many cases. Additional control messages, known as *null messages* are passed along the channels to overcome this problem. A null message with timestamp t is used to indicate that no message with a timestamp less than t will be sent along the channel in the remainder of the simulation. In order to determine the timestamp to give to a null message, the sending LP uses *lookahead*. The lookahead on a channel represents the minimum timestamp increment that could be given to the next event passed along the channel. The timestamp increment is the difference between the timestamp of an event and the timestamp of the event that caused it to be sent.

The simplest form of lookahead is the minimum time increment that could be given to an event. More sophisticated lookahead techniques are possible, such as the pre-computed service time scheme presented by Dickens *et al.* in [14]. The important point is that lookahead is application dependent and is difficult to hide from the simulation modeller.

Conservative simulators can suffer from poor performance if there is little lookahead in the model, or if there are cycles in the communication graph described by the channels between LPs. If there is little lookahead in a cycle of channels and the event density in the cycle is low, many null messages will need to be sent for each event executed at an LP in the cycle. Some algorithms exist that can reduce the number of null messages required, Wood and Turner [73], and Blanchard *et al.* [5], but many null messages will still be required in certain cases.

One very interesting feature of the CMB algorithm is that a sequential simulator using this algorithm can perform better than a central event list sequential simulator[1]. This is possible due to the improved cache behaviour of programs using the CMB algorithm.

1.1.3 Optimistic Algorithms

The Time Warp algorithm, Jefferson [32], introduced the idea of optimistic event execution for PDES. An optimistic algorithm will execute an event even if it cannot be proved to be safe to execute. Since this might result in events being executed out of order, the system needs to be able to recover when a causality error has been detected. In Time Warp this recovery process is called *rollback*. Because optimistic algorithms exhibit risk, a method is also required to cancel event messages generated in error. This task is accomplished by sending *anti-messages*. An anti-message contains the same system state as the message it is being used to cancel except for one field marking it as an anti-message. When the anti-message finds its event message pair, both messages are removed from the system. This process is referred to *event annihilation*.

The need for this recovery mechanism makes optimistic simulation systems more complicated than conservative simulation systems, but in terms of usability, there are some major advantages to using an optimistic approach. There is no need to pre-define communication channels between LPs, so the modelling application programmers interface (API) to an optimistic system can look very similar to the API of a central event list based sequential simulator. Also, an optimistic simulator does not require lookahead information in order to operate.

An optimistic simulator can take advantage of parallelism that conservative systems cannot. An example of this is where it is possible for an event to be sent from one LP to another, but in practice no event is sent. In a conservative simulator the receiving LP would have to wait until it is known that the event message will not be sent; in an optimistic system the receiving LP is free to continue processing future events.

[1] This information was first relayed to me by David Bruce and Chris Booth at FCRC in 1996. This behaviour has been reported by several other researchers since then.

On problem with Time Warp systems is that they require the modeller to provide hooks for the recovery mechanism employed during a rollback. Ways of hiding the recovery mechanism from the modeller are considered in chapter 3. Another problem is that optimistic simulators can suffer from unstable behaviour patterns, that can lead to poor performance; this issue is addressed in chapters 6, 7 and 8.

1.1.4 ANR Algorithms

ANR algorithms are mainly used for man-in-the-loop interactive simulations. The best known ANR algorithm is Breathing Time Buckets (BTB), Steinman [65], as used in the SPEEDES simulation system, Steinman [64]. ANR algorithms tend not to be efficient for general purpose simulators as they cannot take advantage of as much parallelism as optimistic algorithms are able to for simulations with little lookahead, Bellenot [4]. Despite this, work done on ANR algorithms is relevant to this thesis work and the ANR approach is discussed in chapter 6.

1.2 Thesis Overview

This thesis work started out as a general investigation into the viability of being able to create a general purpose PDES simulation system for use in a distributed memory programming environment. It soon became clear that the Time Warp algorithm would be a good basis for such a system, due to it allowing a similar programming model as used by central event list sequential simulators and not suffering from programming restrictions brought about by low lookahead components in the model.

While thought was given to a wide range of problems of Time Warp, and other PDES schemes, the work presented in this thesis concentrates in two main areas. The first of these is an investigation into providing state saving, a mechanism required by Time Warp, in a way that is transparent to the simulation modeller, i.e., the person using the system to write simulations. The second area is that of event scheduling, with the aim of this investigation to produce an algorithm that avoided the worst case behaviour that Time Warp systems can experience, while not unduly effecting the performance of simulations that would not have experienced these unfavourable modes of behaviour anyway. A problem that was encountered early in this work was that the information supplied by the *global virtual time* (GVT) calculation system needed to arrive at each executive in a dependable manner and this was not achieved by the algorithm implemented originally. This led to work on creating a GVT calculation system that achieved this goal. An important aim of the event scheduling algorithm was to meet the scheduling goals without requiring the simulation modeller to tune the algorithm for a particular simulation.

Chapter 2 starts by explaining the Time Warp system written to test the ideas presented in this thesis. It is also used to introduce terms that will be used in the remainder of the thesis.

The transparent state saving system is then presented in chapter 3. Also presented are brief explanations of the different state saving techniques that could be used, along with insights gained from implementing the system presented.

The rest of the thesis concentrates on subjects relevant to event scheduling algorithms; this includes the work done on GVT calculation. In chapter 4, the behaviour of a Time Warp system, along with the factors that define this behaviour are explored. This is important since the primary purpose of the event scheduling algorithms is to optimise the performance of the system as a whole by controlling the onset of inefficient modes of behaviour.

Chapter 5 describes the GVT calculation algorithm that was designed to give predictable GVT updates that where found to be required by some of the event scheduling algorithms that were explored. Then chapter 6 explains the event scheduling problem and presents briefly previous work in this field.

Chapters 7 and 8 then explain first the theory behind, and then the implementation of the event scheduling algorithm that has resulted from this thesis work. Then test results reporting the performance of the Time Warp system, and more particularly the new scheduling algorithm are presented in chapter 9, with some of the performance graphs appearing in Appendix A. The thesis is summarised and closing remarks are given in chapter 10.

Chapter 2

Distributed Time Warp

This chapter describes the Time Warp system developed for this thesis work while introducing terms used to describe aspects of a Time Warp implementation. The system, called *Distributed Time Warp* (DTW), uses a message passing programming model and is therefore capable of running on any multiple instruction multiple data (MIMD) computer with suitable communication library support. This includes dedicated distributed memory massively parallel processor (MPP) computers, clusters of workstations connected via a high speed network as well as shared memory symmetric multi-processor (SMP) and non-uniform memory access (NUMA) computers.

The Time Warp algorithm was introduced in section 1.1.3. This section explains Time Warp in greater detail with an emphasis on the algorithms used by DTW. A Time Warp system has a number of interacting algorithmic components. Each of these components could use a number of different algorithms and be implemented in many different ways. This leads to different Time Warp systems actually operating in quite different ways. The primary purpose of this chapter is to make clear how DTW has been implemented.

The chapter starts by explaining the basic modelling technique used in DTW, its representation of time and defines the meaning of the different clocks used in the system. It continues by describing the structure of DTW and the objects and algorithms employed. An important part of this structure is the multi-LP model explained in section 2.9. The chapter is summarised in section 2.13.

2.1 LP Model and Virtual Time

The system modelled in the simulator is represented using the *logical process modelling methodology* described in section 1.1. With this modelling technique the physical system is represented in the simulator by a set of *logical processes* (LPs) that interact by exchanging event

messages. The LPs and event messages are represented by C++ objects in DTW.

In this thesis simulation time will be referred to as *virtual time* as done by Jefferson in [32]. DTW uses a simple floating point representation of virtual time. In general a simulation starts at a virtual time of zero and terminates when there are no events left representing actions in the physical system occurring before the user defined simulation end time. It should be noted that while DTW shares this simple time representation with systems such as *GTW*, Fujimoto [20], *WarpKit*, Xiao and Unger [74], and *Warped*, Martin *et al.* [39], an extended representation of virtual time was used in *TWOS*, Jefferson [29]. In the TWOS representation of virtual time, each event timestamp value is extended with the identity of the LP that generated the event and with an additional counter. These extensions allow a simulation to run deterministically, both sequentially and in parallel, even when equal timestamped events arrive at the same LP. With some models an extended representation can be useful, but in general deterministic results are not vital. In cases where they are, a minor change in the way the system is modelled can often assure deterministic results without an extended virtual time representation.

The *local virtual time* (LVT) of an LP is the value of the timestamp of the last event message to begin execution at the LP. Any event message in the system that is currently being executed, or that has to be executed in order for the simulation to complete is called an *active* event message. The term *local queue time* (LQT) is used in this thesis to refer to the timestamp of the next event waiting to be executed at an LP. If there is no such event then the LQT for this LP is set to ∞. I use the term *executive virtual time* (EVT) to describe the minimum active message timestamp local to the executive. This is the minimum of the local LPs LQT values and of the timestamps of any events currently being executed. The *global virtual time* (GVT) is a lower bound on the timestamps of all active messages in the system. Note that since any new event message generated will have a timestamp greater than or equal to that of the event that caused it to be generated, no message with a timestamp smaller than the current GVT will be executed in the remainder of the simulation.

DTW works asynchronously with no executive ever waiting for another executive. It does not have synchronous GVT cycles where every executive stops and waits for a new GVT value to be calculated. Partly due to this, I have found using GVT as the only descriptive measure of the passage of virtual time in the entire system to be inadequate. An additional term called the *System Base Time* (SBT) is used to describe the minimum active timestamp at any wall clock time. Therefore, while the simulation is running, the SBT is the minimum of:

i) The EVT values of all the Time Warp executives.

ii) The timestamps of all the transient messages in the system.

I define GVT to be the most recently calculated lower bound on the SBT. The reason for introducing the concept of a system base time is to clarify statements made about the state of

the system. If GVT has the definition given to SBT here, it can be difficult for the reader to determine if the value described as GVT refers to the value of the minimum active timestamp in the system, or to the last value that was calculated. In this thesis GVT always refers to a calculated value that can only be a lower bound on the SBT while the system is running. Also note that in general it will not be possible to actually calculate the SBT, it may be possible to estimate its value however.

2.2 DTW Structure

The DTW simulator is constructed as a set of program executives called *Time Warp Executives* (TWEs), that communicate via the computers message transport layer (MTL). The MTL could be a high speed network, or a library built on top of a shared address space. Currently DTW uses the MPI [2] message passing API making the system very portable. Each TWE is allocated to its own processing element (PE). Allocating more than one executive to a PE is likely to result in lower efficiency due to the usual problems of poor cache locality and context switching cost, but also due to decreasing the message passing locality for LPs since less LPs will be allocated to each TWE. With decreased message passing locality, more events are passed via the MTL. This adds overhead to the sending of the event message and increases the chance of rollbacks occurring; this issue is discussed in section 4.4.

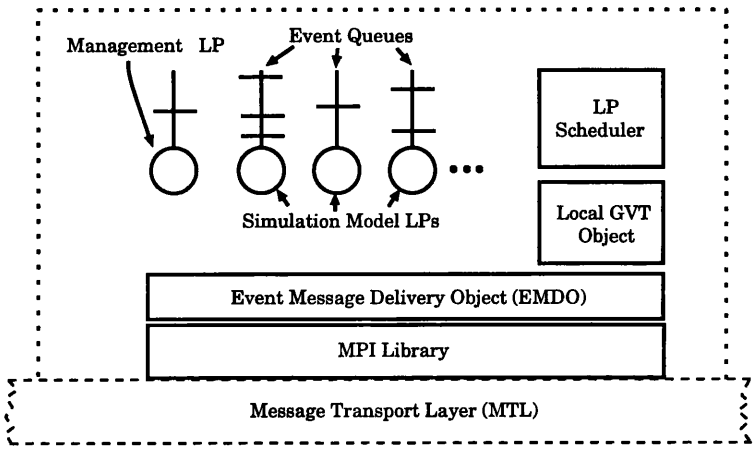

Figure 2-1: Basic elements of a Time Warp executive.

A diagram showing the components of each TWE is presented in figure [2-1]. Each TWE has an LP Scheduler, an Event Message Delivery object (EMDO), a local GVT object, a management LP and the LPs representing the physical processes being modelled. All of these components will be explained in this chapter. As can be seen in the diagram the EMDO uses

MPI to communicate with the computers MTL.

Apart from the TWEs an executive is needed to perform the atomic calculation of new GVT values. This could either be performed on an additional executive whose sole purpose is to perform this calculation, or it could be performed by a GVT calculation object placed on one of the TWEs. This issue is discussed in chapter 5.

2.2.1 Logical Processes

The logical processes (LPs) are represented as C++ objects in DTW. Each LP has its own event queue and its own state manager that manages the LP's old state values and executed events that may be required in the case of a rollback.

Event Queue

The LP event queues are implemented using heap data structures, Sedgewick [51]. The vector used in the representation grows and shrinks automatically. While a *calendar queue*, Brown [6], can give better performance in a system that has one queue per executive, they can be difficult to tune. Event queues based on *splay trees*, Sleator and Tarjan [53], can be effective when events often need to be removed from long event queues. Since the event queues in DTW only have to hold the future events for a single LP, the efficiency advantage that could be gained by using either of these other data structures should be small.

Event cancellation can be difficult using heaps since a tree traversal is required to find an element with a particular key. A recursive breadth first search is used to find the event to be cancelled. Once it is found, this event is cancelled and the event message in the last position in the heap (call this msg_l) is placed in the position previously held by the cancelled event. Then msg_l is bubbled down the heap as necessary. This is done using the standard heap operations, i.e., if msg_l is at position i in the heap, its timestamp is compared with the timestamps of the events at positions $2i$ and $2i + 1$. If the timestamp of msg_l is greater than the timestamp of either of these events, msg_l is exchanged with the event with the smaller timestamp. This continues until either msg_l's timestamp is smaller than the timestamps of the events at positions $2i$ and $2i + 1$ (where i is always set to msg_l's current position), or until the end of the heap is reached. As soon as a message is cancelled, any event messages that were generated directly as a result of the cancelled event's execution are cancelled immediately. This is done so that false messages are removed from the system as quickly as possible.

State Manager

The state managers employ incremental state saving. The implementation of the state saving mechanisms are explained in chapter 3.

2.2.2 LP Scheduler

Each TWE has a single LP scheduler; see figure [2-1]. DTW supports a number of different scheduling policies which can be selected as a compile time option. The default scheduling algorithm is *Lowest Timestamp First* (LTF) scheduling. With this, the LPs are kept in a priority queue sorted by their LQT values. The priority queue is implemented as a heap data structure in DTW. With LTF scheduling, if the LP at the head of the queue has an LQT value less than ∞, then it is scheduled to execute one event. After this it is reinserted into the heap structure using its new LQT value as the sort key. The heap data structure has to be resorted if a new event arrives for an LP that has a timestamp less than the LPs current LQT. In this case the LP is bubbled up the heap if necessary. This is done be comparing the LQT of the updated LP, which is at position i in the heap, with the LVT of the LP at position $\frac{i}{2}$. If the LQT of the LP at position i is less than the LQT of the LP at position $\frac{i}{2}$, the LPs are exchanged and a new test is performed on the LQTs of the LPs held at positions $\frac{i}{2}$ and $\frac{i}{4}$. This continues until either the updated LP reaches the top of the heap, or its LQT is greater than that of the LP with which a comparison is being made.

Note that DTW requires an LP scheduler, since it does not use a single event queue per executive. If a single event queue is used, then LPs can simply be scheduled when an event for that LP is removed from the head of the event queue. This is good if LTF scheduling is being used, but is not so effective if some other event scheduling polices are being employed. Some different event scheduling algorithms are presented in chapters 6 and 7.

2.3 Event Messages

When referring to events in DTW the terms *event*, *message* and *event message* can be used interchangeably. This is because the event messages passed between the LPs are queued and then executed as events. In a system such as TWOS, Jefferson [29], the distinction is made between messages, used to pass information between different LPs, and events that trigger actions at LPs. In TWOS, the arrival of a message will schedule an event at the receiving LP, unless an event has already been scheduled with the same timestamp, in which case the new message is added to that event structure. The TWOS method can eliminate some modelling problems related simultaneous events [72] (i.e., events that arrive at the same LP with the same timestamp), but does have addition overhead.

Event messages are represented by C++ objects in DTW. Each message object has two sections; a header, containing the information needed by the system to correctly deliver the message and the body or text of the message containing the information to be processed by the LP during the execution of the event.

The event message header contains the value of the virtual time at which the action rep-

resented by the event message will occur. This time value is referred to as the message's *timestamp*. The timestamp is the only time value that the message needs to contain in order for the system to execute events in the correct order. Some algorithms that can be employed within a Time Warp system also require knowledge of the timestamp of the event that caused an event message to be generated, so in DTW the message objects contain two virtual time fields. The *virtual send time* (VST), representing the timestamp of the event that generated the message and, the *virtual receive time* (VRT), which represents the time that the event represented by the message occurs. Note that VRT and message timestamp both refer to the same value.

Time Warp uses *anti-messages* to cancel event messages that have been found to have been generated incorrectly. In DTW an anti-message is represented by a message object with an empty body. The message object header contains the same information as the header of the event message it is meant to cancel except for one field, the sign field, which is set to positive in header of an event message and negative in the header of the anti-message.

2.4 Message Delivery

Each TWE has an *Event Message Delivery Object* (EMDO) (see figure [2-1]). This handles the delivery of event messages that are passed between LPs; *self* event messages (i.e., events sent from an LP to itself) are inserted directly into the LP's queue without using the EMDO. All non-self events generated locally to the TWE, as well as all event messages arriving from the message transport layer (MTL) are handled by the EMDO. An event that is passed between LPs on the same executive is called a *local* event message. An event that is passed to an LP on another executive is called an *external* event message.

The EMDO operates by reading the destination LP identity from the message header and using this to find the identity of the TWE to which the LP is assigned from a table. If the destination LP is local, the EMDO inserts the message into the receiver's message queue. Otherwise the EMDO packs the message into an inter-processor message buffer and passes the buffer to the MTL along with the identity of the TWE holding the receiver LP. The MTL then transfers the message to the EMDO on the receiver's host TWE, where it is read by the local EMDO and inserted into the destination LP's event queue.

In DTW a set of parameterised types is provided for message object slot values. This allows message objects constructed from these types to be copied to and from the MTL automatically. For complex types the base class that is used by the parameterised type classes can be used to define classes of objects that can be written to and read from the MTL automatically. An example of this is an object that holds a list of objects of a particular type that allows the list of objects to be written to and read from the MTL along with the simple slot types such as integers held in parameterised type objects.

A problem with this scheme is that it requires message objects to be stepped through to both write and read the data. This is less efficient than copying a contiguous block of memory, as is done in Warped, Martin *et al.* [39]. The advantage it gives is that it allows complex message objects (e.g. ones containing pointers) without requiring the modeller to write specific methods to read and write each message type.

Any event that is sent with a VRT greater than the simulation end time is discarded and is not placed into an event queue or sent via the MTL. This means that when the simulation end time is reached there are no more events to execute and the simulation terminates.

2.5 Rollback and Fossil Collection

A Time Warp system must be able to recover when the local causality constraint has been found to have been violated or when it is discovered that a false event has been executed. The recovery is achieved using the rollback mechanism. During a rollback old state values are restored to the LP and previously executed event messages reinserted into the LP's event queue. A rollback will only occur on the arrival of a message with a VRT lower than the LP's LVT, or when an anti-message arrives. An active message that has a VRT lower than the LVT of its destination LP is described as a *straggler.*

Once GVT passes the VRT of an event, the event is said to be *committed.* Since no event message can arrive with a VRT lower than GVT, no rollback can occur to before GVT. Due to this, none of the state saved before GVT or events timestamped before GVT will be required by any rollback that could occur. This allows the memory used to hold these state values and these events to be released. The process of removing old state that is known not to be needed any longer is referred to as *fossil collection.* Without fossil collection, the executives would soon run out of memory, causing the simulation to fail.

2.6 Memory Management

In DTW the memory used by all event message objects on a TWE is allocated from a single heap dedicated to this purpose. All message memory blocks are the same size even if messages of different types have different object sizes. The maximum size of an event object has to be set by the simulation modeller before calling the function that starts the simulation initialisation phase (see section 2.7). The memory blocks are maintained in a simple linked structure making allocation and deallocation of event message objects computationally inexpensive.

When a message is sent, a copy of the message is kept by the sending LP. This is different from some other Time Warp implementations where when a message is sent the anti-message that could be used to annihilate the message is inserted into the sent message list, Fujimo-

to [20]. In DTW messages are treated as read only objects once they are constructed. If a message is sent to an LP on the same executive as the sending LP, a single message object is shared between the two LPs. A simple reference counting scheme is used to prevent a message record being garbage collected before both LPs have finished with it. An anti-message does not contain the text of the original message, only the header which is identical to that of the message being cancelled, but with the sign field set to negative.

2.7 Execution Phases

The simulation modeller using DTW is responsible for writing the main function for the simulator. In this they have to call their own function construct LPs and set the maximum event message size. Then they have to call the *start_simulation* function. Once this function is called, DTW performs an initialisation stage before the simulation execution phase begins. When the simulation execution phase is complete, DTW enters a termination phase in which statistics can be collected.

During the *initialisation* phase the *initialise* method is called on each LP. When the initialisation phase is complete, all the events with zero VRTs currently in the system are executed by their destination LP. A termination detection algorithm linked to the GVT calculation algorithm (see chapter 5) is used to determine when all of the events with zero VRTs have been executed. When the *initialisation* phase is complete, all the executives are signalled by means of an inter-process control message and move to the *simulation* phase of the run. In this phase events are executed using Time Warp to assure that the local causality constraint is maintained. The *simulation* phase completes when there are no events left to execute. Note that since event messages with VRTs greater than the simulation end time are discarded when they are sent, there are no events in the system when the end time is reached. When the completion of the execution phase is detected, the system moves to the *termination* phase, and the *terminate* method is called on each LP. This allows for final statistics to be collected.

2.8 Lazy Algorithms

An important modification to the original Time Warp algorithm adds laziness to some of the underlying algorithms, West [70]. A lazy algorithm puts off performing some action until the result of the action is required. The hope is that by the time the result of the action is required, the result will have been proved not to have been required after all, so the action is never actually performed.

In a system using *lazy cancellation* no anti-messages are generated until the positive message that will be cancelled by the anti-message has been proved to be incorrect. This is in

contrast to *aggressive cancellation* where anti-messages are sent for all messages generated by events that are rolled back. In a system using *lazy reevaluation* state changes are only recomputed after a rollback if the processing of the event causing the rollback changed the state of the receiving LP. This is in contrast to *aggressive reevaluation* where all state changes are recomputed after a rollback.

Time Warp systems using lazy cancellation can perform much better than systems using the original aggressive algorithm, Bellenot [4]. DTW has both lazy and aggressive cancellation. These are selected using a system wide configuration option. The merits of using Lazy cancellation are discussed in section 4.3. DTW does not use lazy reevaluation. A lazy reevaluation implementation would be quite complex and I know of no working lazy reevaluation implementation.

2.9 Multi-LP Model

In DTW, many LPs are allocated to each TWE. I describe this as the *multi-LP* model. Not all Time Warp systems allocate multiple LPs to each executive; in some only one LP is allocated to each executive. In these systems the physical system being modelled is partitioned into n physical processes, where n is the number of PEs available. I describe this as the *single-LP* model.

There are several reasons why DTW does not use the single-LP model. Strictly in terms of modelling it is usually easiest to model a physical system in terms of its simplest components. In a large physical system this results in a large number of physical processes needing representation, likely far more than the number of PEs that will be used. The second reason is that LPs can be seen as units of parallelism since the synchronisation mechanism only requires that the local causality constraint is observed at each LP to assure correctness. By partitioning the same physical system into more physical processes, it is possible that more parallelism could be exposed.

In DTW each LP has its own event queue and its own saved state queue. It is highly desirable for a system using the multi-LP model to have separate state queues for each LP since it allows LPs on the same executive to rollback independently of one another. The use of independent event queues is not a requirement for a system using the multi-LP model. However, it does allow for some event scheduling policies to be implemented that would be difficult using a single event queue per executive. One such scheme is the *Belief Based Scheduling* (BBS) algorithm described in chapter 7.

2.10 Input/Output

I/O has to be handled carefully when Time Warp is used. In the case of output once information has been displayed on a screen or sent to a printer it cannot be rolled back. Therefore, anything output from the system must be known to be correct. In the case of input, without special mechanisms to handle rollback, information read could be lost during a rollback.

DTW gives the user a simple causally correct output method. This is semantically equivalent to the C library function *fprintf*. Instead of outputting the data immediately, the output request is packed in an event structure with both VST and VRT set to the LVT of the LP calling the output function; at this time the LVT of the LP is equal to the VRT of the event being executed. This output event is queued at the system management LP (see figure [2-1]) and is output to the required file stream when GVT passes the event's VRT. Note that by using an event, the output request will be handled in a causally correct manner using the same mechanism that maintains the local causality constraint for the simulation events, with output events being cancelled by anti-messages before being executed if required.

DTW does not have special support for handling input in a causally correct manner. Handling read only files would not be difficult, but support for interactive user input would require substantial modifications to the system resulting in a large additional system overhead, Ghosh *et al.* [24].

2.11 Other Algorithmic Issues

DTW does not support event preemption. Event preemption may be beneficial for simulations with very large event granularities, i.e., where a large amount of computation is performed in the execution of an event. In this case a new event may wait for a long period of time while the current event is executing, even though the new event proves that the current event is being executed out of timestamp order. Ideally the arrival of this new event should preempt the execution of the current event causing its execution session to be aborted, but such a scheme would add additional overhead to simulations where it is not required. As it is, messages are only read from the MTL between event executions.

Event preemption has a more important rôle in that it can prevent the simulator from failing to terminate due to an event execution session entering an infinite loop due to inconsistent information delivered by a false or early event. These problems can be avoided by "careful" programming, but this requirement is not consistent with the aims of this thesis.

DTW does not enforce flow control. Flow control is used to control the amount of memory required by each TWE. The best known method is *cancelback*, Jefferson [30], which operates by returning high timestamped event messages to the LPs that sent them thus causing these LPs

to roll back. The cancelback algorithm is difficult to implement even using a shared memory programming model; a distributed implementation was felt to be beyond the scope of this thesis work.

2.12 Why DTW?

At the point when this thesis work began there were no Time Warp systems available in the public domain. The first version of the Warped system, Martin *et al.* [39], was released during the first year of the work, but this did not provide a good basis for the particular areas I was looking at. In particular, it did not have incremental state saving and only used one event queue per executive. Since DTW was already working before Warped was released, these difficulties made me stay with DTW as my development system.

2.13 Summary

This chapter has introduced the structure of and algorithms used by the Distributed Time Warp (DTW) system. It has also introduced terminology that will be used in the remainder of this thesis.

One of the key points described is the use of a multi-LP model with independent state managers and event queues at each LP. This allows event scheduling policies to be employed that could be difficult to implement efficiently if shared event queues or shared state managers were used. Indeed, the event scheduling algorithm developed in chapter 7 of this thesis could not have been implemented efficiently if a different event queue and state manager configuration was used.

The state management and GVT calculation schemes were introduced, though full explanations of these parts of the system were left for later chapters. The important points are that incremental state saving is employed, and that GVT is calculated asynchronously without the simulator having to stop while the calculation is performed.

Specific details of some parts of the system will be given in chapters that follow. The state management scheme is described in chapter 3. Then the GVT calculation algorithm used, along with details of the GVT calculation subsystem are described in chapter 5. Finally, event scheduling policies are discussed in chapter 6 and a new event scheduling policy to replace the LTF scheme presented in this chapter is developed in chapter 7.

Chapter 3

State Saving

The optimistic execution policy employed in Time Warp necessitates a mechanism being available to restore old state values when a rollback occurs. To achieve this, either old state values have to be maintained so that they can copied back, or reversible code sequences have to be maintained so that the old values of state variables can be calculated when required.

The purpose of this chapter is to explain the automatic incremental state saving system implemented in DTW. It starts by explaining the different techniques that have been used for state saving in the past. Then the problems that can be caused by simple programming errors in the modellers state saving code are examined in section 3.2; these are introduced as part of the motivation for a transparent state saving implementation. Section 3.3 explains how state saving extensions can be added to C++ to provide automatic state saving and thus eliminate the problems discussed in section 3.2. Sections 3.4 and 3.5 explain the state saving and recovery methods used in the scheme and point out some efficiency problems. A brief discussion on the effectiveness of the transparent state saving scheme is given in section 3.6 and the chapter summarised in section 3.7.

3.1 State Saving Methods

To date, all Time Warp systems have used state value copying to implement the recovery process required when a rollback occurs. It was suggested by Gomes [26] that the recovery process could also be accomplished using code that reverses the actions of code run in the forward execution phase. While this is an interesting idea, it will not work for all situations where state needs to be saved. For example, many floating point operations would be difficult to reverse. Also the requirement to write code that reverses the actions of the forward execution code, either by hand or using a compiler, makes this unsuitable for the purposes of this thesis. Therefore, this chapter concentrates on state value copying methods of state recovery.

There are two basic ways of implementing value copying state saving in a Time Warp system, *copy state saving* (CSS) and *incremental state saving* (ISS). These are described in the following two sections.

3.1.1 Copy State Saving

With *copy state saving* (CSS) the entire state of an LP is saved. CSS was employed in the first Time Warp system to be implemented, Jefferson [32]. The act of copying the state of an LP is performed either before each event is executed or after some number of events have been executed. This second scheme is called *periodic checkpointing*, Lin *et al.* [34] and is done to try to reduce the overall state saving overhead.

If the LP is large, the amount of time it will take to save its state will also be large. If the amount of state updated by the processing of an event is only a small part of the LP's state, copying the entire LP before each event is processed wastes a large number of CPU cycles.

3.1.2 Incremental State Saving

With *incremental state saving* (ISS) the value of a state variable is only saved when the value of that variable is about to be updated. When an ISS LP is rolled back, all old state values saved since the rollback time have to be restored. This might mean that one variable has its value restored many times in a single rollback. With ISS the time taken to rollback an LP is proportional to the number of state values saved since the rollback time.

One of the first PDES systems to use ISS was SPEEDES, Steinman [63]. In this, the user provided slots in the event structures for the values of any state variables that could be updated by the execution of the event. When the values of the variables are first updated during the execution of the event, the old state values are copied to their place in the event structure. This way each LP only has to maintain a list of executed events with all the saved state held in the event structures.

A different ISS scheme was presented by Cleary *et al.* in [10]. In this, state items are saved to the next free element of a vector and the back-trace consists of a list of one or more state vectors. Each LP has a state vector list and a saved event list. When an event starts execution, the pointer to the next empty element in the saved state vector is set in the event structure. Rollback operates by first rolling back all the events that have been executed early. Then all state from the state vector is copied back until the element of the state vector is found that is pointed to by the pointer value held in the last event rolled back. At this point the LP's state has been correctly restored to the values held before any of the rolled back events were executed. The transparent ISS scheme presented later in this chapter is based on this method.

3.1.3 Comparisons

Both ISS and CSS have advantages and disadvantages; which one will give the best performance will depend on what state is updated. CSS performs poorly if only a small proportion of an LP's state is updated. This situation can be improved by using periodic checkpointing, though this introduces the chance that state will not have been saved at the point in the LP's history that the LP needs to rollback to. In this case the LP has to rollback to the last point previous to the rollback time where state was saved and then re-execute events during a *coast forward* phase, back to the rollback time. During this coast forward phase all events generated must be discarded. If the checkpointing period is too large, the simulation performance will suffer due to the time spent coasting forward.

ISS can perform poorly if a large proportion of an LP's state is saved for each event executed. ISS systems, such as the one presented in [10], can also suffer from performance problems due to poor placement of the state saving instructions that can cause state values to be saved more than once during the execution of a single event. This does not effect the system used in SPEEDES [63], since only one value is saved from any state variable during any event execution. Unfortunately, without complier support the SPEEDES scheme is far from transparent.

For any event that is executed only the value held in a variable when the event execution begins has to be saved. If the same variable is updated multiple times during the execution of an event, saving the contents of the variable each time will have no effect on the value held in the variable after a rollback occurs. This is because all values held in the back trace are copied back, so any values saved later will be overwritten by the value saved first. Saving the contents of a variable multiple times adds both to the forward execution cost and also to the rollback cost. An example of one instance of this can be seen in figure [3-1]. In the code on the left, a state variable is saved on each iteration. In the code on the right, a single state save is made before entering the loop.

```
for( i= 1 ; i < 10 ; i++ ) {
  save_state_var( &s1 );
  s1+= incVAL;
}
```

```
save_state_var( &s1 );
for( i= 1 ; i < 10 ; i++ ) {
  s1+= incVAL;
}
```

Figure 3-1: Removing an incremental state save instruction from a loop to improve the efficiency of the system.

If a large number of functions are called during the execution of an event, the same variable could be updated in many places. Unlike the loop variable case (figure [3-1]) this could be very difficult to notice without extensive analysis.

The decision of which state saving scheme is not simple as it depends greatly on the particular simulation being run. A comparison of periodic checkpointing and ISS can be found in

Palaniswamy and Wilsey [43], and this issue will be discussed further in section 3.6.

3.2 Non-Deterministic Errors

Programming errors in the state saving and state restoration code can be extremely difficult to detect. This is because the runtime errors they cause tend to occur in a non-deterministic manner. Consider the case where a modeller has to write both the state saving and state restoration code for their LP classes. Suppose that the modeller fails to state save one of the state variables in an LP class. The runtime errors caused by this programming error will occur non-deterministically depending on when any instance of the LP class rolls back. Other programming errors of this type include, saving the value of a variable but failing to restore it, or saving the value and restoring it in the wrong location.

To eliminate programming errors in the state saving code, the cause of the errors needs to be removed. Since it is the modeller's state saving code that will introduce the errors, state saving must be implemented transparently by the system.

3.3 Automatic Incremental State Saving

The automatic state saving code developed in this section was developed independently based on ideas for incremental state saving presented by Cleary *et al.* in [10]. Since being developed, two other implementations of automatic incremental state saving using similar techniques have been published by Ronngren *et al.* [50] and by Gomes *et al.* [25]. The most important difference between the scheme presented here and the other schemes, is that this scheme enables items from lists to be state saved.

The main reason that ISS is used is that a transparent state saving scheme can be incorporated into parameterised types in the general purpose programming language C++. While CSS may offer advantages in some cases, implementing CSS transparently would either require placing strong restrictions on the structure of LP objects, or require compiler support to provide the state saving and recovery code for particular LP classes.

In DTW, state saving is performed automatically on objects constructed with the parameterised type classes described below. LPs whose state variables are instances of these classes will operate without any specialised state saving code having to be added by the simulation modeller. All the state saving classes are subclasses of the C++ class *StateObj*. The following automatic state saving classes are provided:

- **StateData< dtype >** : where dtype is a standard C/C++ type (e.g. int, double, etc.). This acts as an automatic state saving replacement for the standard C/C++ types.

- **StatePointer< so_class >** : where so_class is any subclass of class StateObj. This provides a pointer class that allows garbage collection to be handled in a rollback safe manner.

- **StateList< so_class >** : where so_class is any subclass of StateObj. This provides a list class with automatic state saving. Elements removed from a state saving list are restored to the position in the list that they held before being removed. This means that lists sorted into some order by the simulation level code are restored in the correct order, without the simulation programmer having to supply an ordering function for re-inserting list elements during rollback.

Other specialised state saving data types can be constructed by subclassing the StateObj class; though the implementor of the object classes must provide the state saving and restoration method required by the class. All of these classes use operator overloading to provide "transparent" state saving; these are not explained here since they are similar to those presented by Ronngren *et al.* in [50].

As shown in figure [2-1] in chapter 2, each LP has a state manager. Each state manager contains a state vector into which values being saved are copied. Each time a state variable is updated in an LP, the state value being replaced is copied to the state vector in the state manager associated with that LP.

Each element of the state vector has four components; a tag indicating the type of data saved in this element, the saved value, an address component and a component containing the size of the data. The meaning of the type field and the address field depends on the type of data saved; this will be explained later. The size field is required since the data field, which is the size of a double type, could be holding a short integer that will be half as long as a double in most operating systems. The start of each data field is aligned using the machines floating point alignment restrictions.

Another option could have been to always store a word of data at a time and hold values that are longer than a word using multiple vector elements; this was done in the state saving scheme presented by West and Panesar [71]. The relative efficiency of these two methods is likely to depend on the number of multiple word saves that have to be made.

The type field is used to indicate how the state item held in a state vector element should be copied back to the LP from which it came. When a state item is copied back, a switch statement using the type field as a key is used to call the correct restoration method. Another option would have been to hold a function pointer to a function that would do the restoration instead of holding the type field. This may be interesting to experiment with, but this has not been done in DTW.

Each state manager maintains a pointer called the *top pointer*. The top pointer points to the

current top of the state vector. When an event is executed at the LP owning the state manager holding the vector, the value of the *top pointer* is copied to the event object; this could be required in the case of a rollback or during fossil collection. If a state value is saved, it is copied to the element pointed at by the top pointer and then the top pointer is incremented.

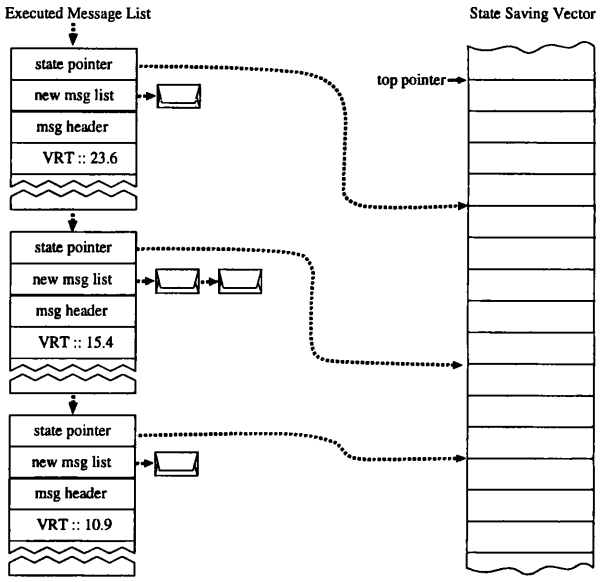

Figure 3-2: The processed message list and state vector.

Figure [3-2] shows how elements of the state vector are referenced from event message objects representing events that have been executed by an LP. Note that the events also hold a list of the event messages generated during their execution; this makes cancelling events easy if an event is annihilated by an anti-message. In this case all the messages that were generated by the event being annihilated can have anti-messages dispatched immediately simply by cancelling each message in the list.

Consider the case where the an anti-message arrives for the message with the VRT of 15.4 in figure [3-2]. First the message timestamped with a VRT of 23.6 will be reinserted into the LPs event queue. If aggressive cancellation is being used the message in this events new message list would be cancelled at this point. Next, the event with the VRT of 15.4 is removed from the executed message list. Since this event is being annihilated the two messages in its new message list have anti-messages sent to cancel them. Then the state pointer is recovered from the old event and all state held in the vector down to the element pointed to is restored to the LP.

In reality, the state managers hold a list of state vectors and tests have to be performed to determine if a new vector is required at the head of the list or if an old vector can be discarded

from the end of the list. Tests also have to be performed to prevent running over the end of a vector when restoring state.

3.4 Saving and Restoring Data

This section explains how state items based on the parameterised classes listed in section 3.3 are state saved and have their values restored. This is explained individually for each of the classes.

3.4.1 StateData< dtype > State Objects

When an object of class **StateData< dtype >** is saved, the element of the saved state vector referring to the object is constructed as follows. A **StdDATA** tag is placed in the type component. The value of the variable is copied to the data field using the C function *memcpy*. The address of the variable is placed in the address component and the size of the variable is placed in the size component. The size of the data type is determined using the sizeof operator in the parameterised function that performs the state saving operation. The state saving function that implements these actions is called from the overloaded operators $(=, ++, --)$ defined in the parameterised class.

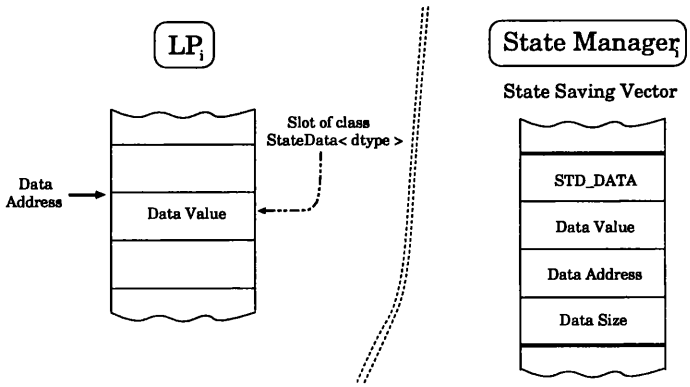

Figure 3-3: A slot of type **StateData< dtype >** in an LP and its saved state representation in the state manager's saved state vector.

State is restored by copying *size* bytes (where *size* is the value held in the size component), of the value held in the data component back to the address held in the address component.

3.4.2 StatePointer< dtype > State Objects

Slots of class **StatePointer< dtype >** are state saved and restored using the same method described for **StateData< dtype >** slots. The **StatePointer< dtype >** is provided to automatically handle fossil collection of the objects pointed to. In order for fossil collection to operate, the pointer must point to a type that is a subclass of *StateObj* which has a virtual destructor which handles the deallocation of the memory that is referenced by the state pointer. In this case, **PtrDATA** is placed in the tag field.

3.4.3 StateList< so_type > Elements

The **StateList** parameterised class is a doubly linked list class that incorporates transparent state saving of the list state, where the list state consists of the elements held in the list and the order of the elements in the list. Note that for garbage collection to work all data elements in the list must be types subclassed from class **StateObj**, as was the case with the objects referenced by the state pointer class. To maintain ordering information, a record regarding every element added to and removed from the list has to maintained in the state vector. A different tag is used depending on whether the element is added to the list (**addListDATA**) or removed from the list (**rmListDATA**) and a pointer kept to the list element in the address field of the vector element.

During a rollback elements tagged with **addListDATA** are removed from the list and garbage collected. Elements tagged with **rmListDATA** are reinserted into the list. List elements are restored in the position in the list they held before being removed. They could simply be re-inserted into the list, but this could destroy any ordering given to the list by the modeller's code. Instead, lists are reconstructed using the **next** and **prev** pointers stored in the list elements; these can be seen in the state saving list shown in figure [3-4].

Since each list element is replaced in the reverse order that they were removed from the list, the **next** and **prev** pointers in the saved list elements refer correctly to the next and previous elements in the list at the time the elements were removed. By changing the **next** pointer in the list element pointed to by the saved element's **prev** pointer to point at the saved element, and changing the **prev** pointer in the list element pointed to by the saved element's **next** pointer to point at the saved element, the list can be reconstructed to the order it was in before the saved element was removed from the list. Similarly, elements referenced from the state vector tagged with **addListDATA** can be removed from the list leaving the list in the state it was in before the element was added.

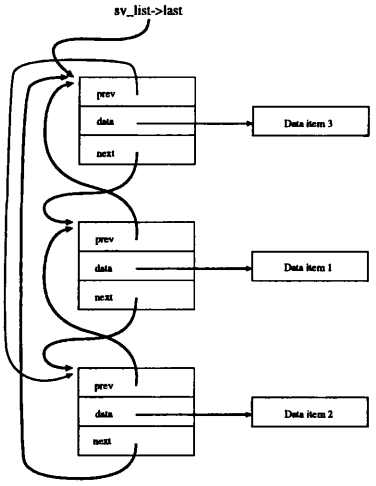

Figure 3-4: A state saving list. These are doubly linked lists that feature automatic incremental state saving and restoration.

3.5 Efficiency Problems

Using the automatic state saving scheme presented in this chapter does have a number of efficiency implications, with the multiple update problem explained in section 3.1.3 being the major one. This problem can be resolved by the modeller using local variables for calculations that would otherwise use state variables and only updating the values of state variables to those of the temporary variables at the end of the event execution. While this solves the efficiency problem it means that the modeller has to understand that state variables have to be handled in a special way, which removes the transparency that was the aim of this work. One transparent solution would be for the system to determine if a state variable had already been state saved during the execution of the current event, and preventing it from being saved again.

Another efficiency problem arises when a state variable is updated with the same value it held before. Restoring this old value will not effect the state of the LP and saving the value will only add to the forward execution and rollback costs. Again an experienced programmer would test for this case and avoid saving a value that will not be updated, but the aim is to produce a scheme that can be used by a non-expert PDES programmer. In this case the system could implement a test to prevent the save from taking place, but this would add to the forward execution cost.

The list state saving implementation has particular efficiency implications, since if a modeller were to sort the list by removing and reinserting the elements, an element would need to be added to the state saving vector both when an element was removed and when it was

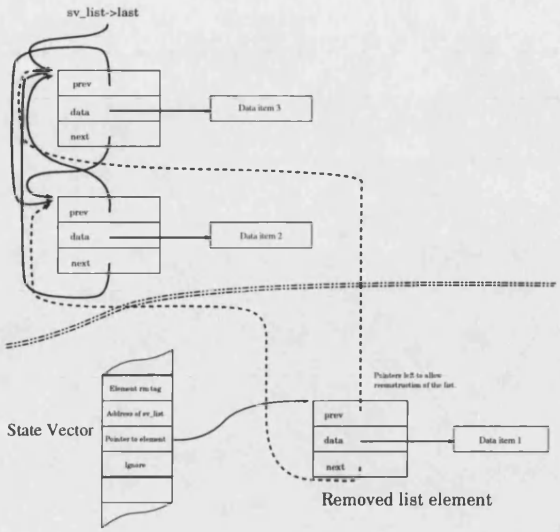

Figure 3-5: Diagram showing an element removed from a state saving list and placed in the state vector. Since the next and prev pointers are maintained with the element, the list can be reconstructed in the same order as it was in before the element was removed.

reinserted into the list. Here again, this class is useful, but its operation needs to be understood in order to avoid using it in ways that will result in poor performance.

3.6 Discussion

Most of what appears in this chapter has been reported elsewhere, with an exception being the method used to state save lists.

Although the state saving extensions presented in this chapter work well, I do not feel they provide a robust solution to the state saving problem. Though I don't have experience in attempting to implement such a scheme with other general purpose languages, I suspect that they will suffer similar problems, though the ability to access raw memory and to perform unsafe (C style) casts in C++ does cause particular problems. It is all too easy for the modeller to cast an object to a different type and therefore cause the incorrect operators to be called on the object. This could cause the state saving system to fail.

A truly transparent state saving method was presented by West and Panesar in [71]. This again uses an incremental state saving system, but adds the state saving annotations using executable editing. This has the great advantage that it allows state saving to be added to compiled binary code, thus getting around the problem that the modeller does not always have access to all the source code that is used in their system. Unfortunately, the runtime performance of this

system is poor due to the code modifications made by the executable editor. Therefore, this does not provide a general solution to the state saving problem.

3.7 Summary

This chapter presented a method of providing transparent incremental state saving in a general purpose computer language. This used parameterised classes in C++ to provide state saving of simple data types and pointers. Also presented was a specialised state saving list class that maintains the ordering of the list across a rollback. This work could have been taken further, but the work presented by Fabian Gomes [26] covered a large amount of the future work that could have been done in this area.

Although the state saving extensions to C++ presented in this chapter work well, their implementation does need to be understood to use them effectively. The work done in experimenting with these ideas has led me to the conclusion that the provision of transparent state saving extensions in general purpose computer languages using standard compilers is far from ideal. I believe that transparent state saving has to be provided using a specialised compiler.

Chapter 4

Describing Time Warp

This chapter considers factors that determine the behaviour of a Time Warp system. It concentrates on issues relating to DTW, i.e., a Time Warp system using a distributed memory programming model, though many of the issues discussed are also relevant for systems using a shared memory programming model.

A Time Warp system works asynchronously, with the order in which actions occur depending on a large number of factors. It is important to understand these factors in order to understand why some simulations run well in a Time Warp system, while others can experience extremely poor performance, with worse than sequential performance in some cases. This understanding is vital for the construction of algorithms that attempt to control the system to prevent inefficient modes of operation from occurring.

The factors considered in this chapter include the physical system being modelled as well the way in which the system model is represented in the simulator. The chapter starts by defining terms used to describe the event/LP interactions that take place in a discrete event simulation system. Then the concept of a "critical path" is described and supercritical speedup is discussed. Finally the "failure modes" that were first described in by Lubachevsky and Weiss [37] are described. These are a set of modes of operation that can cause a Time Warp system to perform poorly, or even fail to terminate.

4.1 Model Interactions

Causal relationships in the physical system being modelled are maintained by executing events in accordance with the local causality constraint in the logical system. The relationships between events can be described in terms of *predecessor* and *antecedent* relationships, Srinivasan and Reynolds [61]. Let *vrt(e)* be the timestamp (virtual receive time) of event *e*.

Definition 3 *Event* e *is the* predecessor *of event* e' *(or* e' *is the successor of* e*) if:* **(i)** e *and* e' *are executed by the same LP,* **(ii)** vrt(e) < vrt(e') *and,* **(iii)** *there is no other event* e'' *at the same LP such that* vrt(e) < vrt(e'') < vrt(e'). *Denote the predecessor of event* e *as* pred(e) *and the successor of event* e *as* succ(e).

Definition 4 *Event* e *is the* antecedent *of event* e' *if the execution of* e *generates* e'. *Denote the antecedent of event* e *as* ante(e).

Note that definition [3] does not mention events with equal timestamps and therefore with just this it is not possible to describe the non-deterministic relationships between equal timestamped events in systems that allow this (see section 2.1). Another classification would be required to talk about these events, but since it is not required for the descriptions that follow this issue will not be pursued.

Events that are scheduled by event messages that are created during the simulations initialisation phase are described as *initial events.* The set of events waiting to be executed at the start of a simulation is referred to as the *initial event set.* Initial events that do not have predecessors are described as *start events.* An event that is not an antecedent or a predecessor of any other event is described as a *final event.*

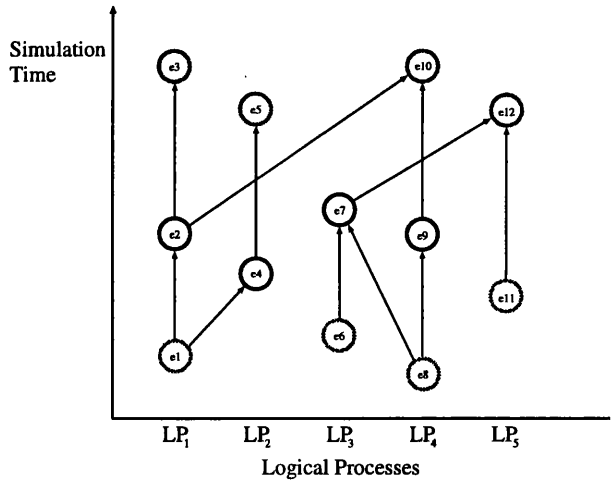

Figure 4-1: A space-time diagram for simple simulation.

The relationships between events are often depicted in the form of a space-time diagram showing the simulation time at which events are executed at each LP as well as the antecedent relationships of events received from other LPs. Figure [4-1] shows the space-time diagram for a very simple simulation where events **e1, e2, e6, e8** and **e11** are initial events. Events **e1, e6, e8** and **e11** are also start events. Events **e3, e5, e10** and **e12** are final events.

A space-time diagram does not differentiate between antecedent and predecessor relationships for events executed by the same LP. For example it is not clear from figure [4-1] if event **e2** is a successor to **e1**, or is an initial event. The space-time view fits in with the local causality constraint view of the system, since events scheduled for execution at the same LP must be executed in timestamp order.

4.2 Critical Paths

Critical path analysis is a technique that can be used to estimate the amount of wall clock time it will take to complete a parallel simulation.

Let **T(e)** be the amount of wall clock time it takes to execute event *e*. The *critical time* of event **e**, **crit(e)** is defined using (4.1), where the simulation is assumed to start at wall clock time zero and **crit(ante(e))** and **crit(pred(e))** are defined to be zero if **ante(e)** and **pred(e)** are not defined respectively, Srinivasan and Reynolds [61].

$$\mathbf{crit(e)} = max\Big\{\mathbf{crit(ante(e))}, \mathbf{crit(pred(e))}\Big\} + \mathbf{T(e)} \tag{4.1}$$

A *critical path* is defined using the following rules:

i) every final event is on a separate critical path.

ii) if **e** is on a critical path and **crit(ante(e))** $\leq$ **crit(pred(e))** then if **pred(e)** exists, it is on the critical path.

iii) if **e** is on a critical path and **crit(pred(e))** $\leq$ **crit(ante(e))** then if **ante(e)** exists, it is on the critical path.

The critical time of a simulation is the greatest critical time of any event in the simulation. The amount of wall clock time that it will actually take to run a simulation will depend on the way in which events are scheduled. A scheduling scheme in which events are executed in their critical path order is called an *elementary scheduling* scheme, Jefferson and Reiher [31]. By definition all conservative algorithms use an elementary scheduling scheme. The original Time Warp algorithm, Jefferson [32], also uses elementary scheduling though this may not be instantly apparent. Although some events are executed early, the work done in executing these events will be undone by the rollback mechanism. The events will then be re-executed obeying the local causality constraint and thus the last time any event is executed, it is executed using elementary scheduling. The amount of wall clock time that it will take to run simulation using an elementary scheduling scheme is bounded below by the simulation's critical time.

4.3 Super-criticality

Suppose that a simulation algorithm is able to execute events out of critical path order and not have to undo all work that is found to have been done early. Then it is possible for this algorithm to complete a simulation in less than the simulation's critical time. One such algorithm is Time Warp with lazy cancellation, West [70], which is used in DTW. With lazy cancellation anti-messages are not sent until it has been proved that the message to be cancelled would not have been generated anyway. This avoids cancelling a message and then soon after sending a new message with exactly the same contents to replace the one that has just been cancelled.

A simulation run that completes in less than the simulation's critical time is said to have exhibited *super-critical speedup*. A scheduling algorithm is described as *super-critical* if it is able to achieve super-critical speedup. Just because an algorithm is super-critical, it does not mean that super-critical speedup will be achieved for every simulation run.

Two events are said to be independent if the result of executing the events is the same whichever order they are executed in. The analysis of super-critical speedup presented by Gunter in [27] came to the conclusion that an event executed early would have to be independent of the straggler causing it to rollback in order to benefit from lazy cancellation. Later it was shown by Srinivasan and Reynolds [61] that super-critical behaviour was possible even if none of the events are independent. In some cases this can occur because the result of executing an event would be the same given a range of LP state values. In other cases, it is because the state variables just happen to have the same values when an event is re-executed, as when it was first executed, even though the event history up to this point is now different.

It should be noted that lazy cancellation has a larger effect on the behaviour of a Time Warp system than simply reducing the number of anti-messages that are generated. In the case where an antecedent of event message **e** is rolled back, if it is later found that **e** was generated in error, the anti-message generated to cancel **e** will have been delayed. This could lead to the side effect of the false messages spreading further than if aggressive cancellation had been used.

4.4 Rollback Cascades

In the original Time Warp algorithm, rollback is the only form of synchronisation between executives. A rollback can occur for one of two reasons; either a straggler message arrives or an anti-message arrives for an event that has already been executed. Note that since an anti-message can only be generated as the result of a rollback, any rollback caused by an anti-message can be traced back to a rollback caused by a straggler. A series of rollbacks triggered by each other is described as a *rollback cascade*, Lubachevsky *et al.* [36].

An interesting feature of the multi-LP model (see section 2.9) is that a message can arrive at an executive with a VRT lower than the VRT of the last event executed on the executive, but not be a straggler and therefore not cause a rollback. As long the event message's VRT is greater than the LVT of the receiving LP, the event is not a straggler. It is possible that a local event **e**, for which the arriving event message is an antecedent could be a straggler however, since it is possible that **e**'s destination LP has advanced further into the future than the receiver of the external event.

The arrival of an anti-message will not necessarily cause a rollback either. If the anti-message's positive counterpart has not been executed by the time the anti-message arrives, or if it has been reinserted into the LP's event queue after a rollback, the pair of event messages can just be annihilated without a rollback.

4.5 Message Lifetimes

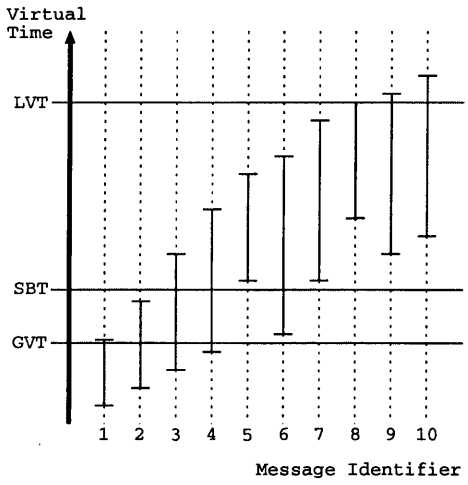

Figure 4-2: A representation of the lifetimes of ten event messages, with a snapshot of the current SBT and the GVT value known to the executive on which the events reside.

The length of virtual time between the VST and VRT of a message is called the message's *lifetime*. The lifetimes of messages affects the simulator in a similar way that the amount of lookahead available affects a conservative simulator (see section 1.1.2). The longer the lifetime of a message, the less likely that its arrival will cause a rollback.

Consider an event **e** that has not yet been committed. Once GVT advances passed **vst(e)**, the risk (see section 1.1.1) associated with **ante(e)** generating **e** is removed, so at this point it is known that **e** will not be cancelled. This fact is used by ANR algorithms to eliminate the need

for anti-messages. Until GVT reaches **vrt(e)** it is not known if all the predecessor events to **e** have arrived.

Figure [4-2] shows the lifetimes of ten events, with a snapshot of the current SBT and the GVT value known locally. Event messages **1**, **2**, **3** and **4** all have VSTs less than GVT; so they are all known to be correct. Event **6**'s VST is less than the SBT, so event **6** is also definitely correct, but since it VST is greater than the value of GVT known by the executive, the executive cannot yet determine that the event is correct. Note that all the events have VRTs greater than GVT. While it can be seen that events **1** and **2** must be correctly ordered, since their VRTs are less than SBT, none of the message are currently known to be correctly ordered by the executive.

If all the events in figure [4-2] where destined for the same LP, then in a sequential simulation, at any point in virtual time, all the messages whose lifetimes span the current virtual time will be in the destination LP's event queue. In a Time Warp simulator, events from other LPs may arrive out of order, so not all the event's whose lifetimes span the LVT of an LP may be in its event queue.

The difference between the VRTs of any two events to arrive at the same LP is described as an *inter-arrival time*. The greater the ratio of message lifetimes to LP event inter-arrival times, the more parallelism is available in the system.

4.6 Self and Local Events

Events can be classified as *self initiating* or *self* events, *local* events or as *external* events. Self events are sent from an LP to itself, local events are sent from an LP to another LP on the same executive and external events are sent from an LP to another LP on a different executive. Notice that if a single-LP model is used, then there are no local events.

Self and local events are known to cause problems for Time Warp synchronisation. This is because an executive may independently execute events far into the future only to have all of these events rolled back on the arrival of an external event. This effect can also occur with external events that are passed between some subset of the executives making up the system. If these executives are executing events with VRTs greater than those being executed at other executives, when an event arrives at one of the fast running executives, this could start a rollback cascade across all the fast running executives rolling back large numbers of events.

4.7 Load Balance

Load balance refers to how work is allocated to different executives making up the parallel system. Load balancing distributed simulations is a difficult problem. If some executives

have more work to do than others, it can lead to thrashing behaviour. This occurs when the executives with more work to do advance their EVTs slower than executives will less work to do; this can lead to the slower executives sending messages that cause long rollbacks on the faster running executives. These long rollbacks can then cause a delayed response from the fast executives, thus leading to them triggering rollback cascades even though they had previously been executing events far ahead of the SBT.

The workload in a simulation can be characterised using the following:

- The rate at which events arrive at an LP.
- The number of CPU cycles used to execute an event.
- The process interaction graph which describes the interactions between LPs.

The best partitioning for a particular simulation may not be obvious. Grouping LPs that communicate often on the same executive can improve the message passing locality which in many cases can improve performance. However, if there are workload hot spots that move around in the model, such a grouping of LPs can lead to work being concentrated at particular executives at particular times. This can lead to the thrashing behaviour described in section 4.8.

Another option is to simply card deal the LPs. Card dealing describes allocating the LP with identity **i** to the executive with identity **i** *mod* **n**, where **n** is the number of executives making up the system. While this gives poor message locality it can even out the effects of hot spots. This has been shown to offer good performance in battlefield simulations where the actions modelled are concentrated at specific physical locations at particular times, Steinman and Wieland [67].

4.8 Failure Modes

A set of situations in which the Time Warp algorithm is either inefficient or fails to terminate were presented by Lubachevsky and Weiss in [37]. The four "failure modes" described are *echo*, *wildfire cascade*, *gushing cascade* and *zero-efficiency simulation.* In order for a parallel simulation to work efficiently these modes of operation should be either minimised or avoided. These modes are of particular interest to the writer of a Time Warp event scheduling algorithms, since one of the aims of such an algorithm has to be to try and prevent the system entering these modes of behaviour.

The analysis in [37] showed that the Time Warp algorithm could be inefficient if false event messages propagate faster than anti-messages can catch up with them. This is why dealing with anti-messages takes priority over dealing with simulation events in DTW (see section 2.2.1).

An *echo* describes a situation where the two or more LPs repeatedly trigger rollbacks at each other with the amplitude of successive rollbacks having an increasing tendency, but where

the number of LPs involved in the rollback cascade does not increase. A *gushing cascade* describes a situation similar to echo, but where the number of LPs involved tends to grow as the length of the rollbacks grow. A *wildfire cascade* describes the situation where the number of rollbacks increases but the size of the rollbacks remains the same. Finally, a *zero efficiency simulation* is a simulation with a finite number of events and a finite number of LPs, but the simulation never terminates as anti-messages are never able to catch up with false messages circulating in the system. This dog chasing its tail situation is also sometimes described as a *vortex.*

Examples of systems that exhibit echo, wildfire cascade and zero efficiency simulation are given in [37].

4.9 Summary

This chapter has explained some of the factors that determine how a Time Warp system behaves when running a particular simulation. The behaviour of a Time Warp system has been shown to be extremely complex.

Many of the factors that determine the behaviour of a Time Warp system are dependent on the interactions between the LPs involved in the simulation. These interactions are in turn dependent on the physical system being modelled and how the logical system used to represent the physical system being simulated is partitioned into LPs.

In chapter 6 the subject of how events should be scheduled on each executive is introduced. Event scheduling algorithms are used to attempt to control the system to prevent unfavourable modes of behaviour, such as the "failure modes" described in this chapter from occurring. From this chapter we see that while it may be possible to control the onset of worst case behaviour using scheduling algorithms, the performance of the simulator will be governed greatly by the system being modelled and by decisions made by the simulation modeller.

Chapter 5

GVT Calculation

The calculation of the *global virtual time* (GVT) is vital to the operation of a Time Warp system. GVT is a lower bound on the timestamp of any active message in the system. Since the lifetime of an event message always has a non-negative value, no event message with a timestamp smaller than GVT will be executed in the remainder of the simulation.

In this chapter an asynchronous distributed GVT algorithm is developed. The algorithm is based on ideas from the *pGVT* algorithm, D'Souza *et al.* [15] and the sequence numbering scheme presented by Lin and Lazowska [33]. The aim of the algorithm is to provide regular GVT updates while limiting the number of control messages needed to achieve this. This work was required in order to provide a GVT subsystem capable of providing GVT information accurate enough to be useful to the scheduling algorithm described in chapter 7.

5.1 Simple GVT Calculation

In this section a simple asynchronous distributed GVT calculation scheme is developed which will be referred to as the *Asynchronous Lower Bound* (ALB) algorithm. The following sections will develop a more sophisticated scheme using ALB as a basis.

This GVT calculation scheme uses a single GVT manager object (GVTman) to perform the GVT value calculation. This GVT manager could reside on one of the Time Warp executives, or operate on an executive dedicated to the task. Where the GVT manager should reside will be discussed in section 5.6. The other components of this scheme are the local GVT managers. These are objects, one of which resides on each Time Warp executive. The local GVT manager (LGVTman) is passed local time information by the event scheduler and sends local time updates to the GVTman. It is the job of the GVTman to collect the reported times sent by the LGVTman objects, and to calculate new GVT values as the minimum of these reported values.

As described by Lin *et al.* [33], there are two problems that have to be overcome in order

to calculate accurate GVT values in a distributed environment. One is that the timestamps of all messages that are in transit between TWEs have to be taken into account; this is described as the *transient message problem.* The second problem is that if different TWEs report their local time information at different wall clock times, the information collected by the GVT manager will not represent a snapshot of the system at a particular wall clock time and may not be consistent. This problem is described as the *simultaneous reporting problem.* To solve the transient message problem a lower bound on the timestamps of all transient messages has to be found. Between when an event message is sent to the transport layer and when it is read by the receiving executive only the sending executive knows the timestamp of the message. Therefore, transient message information has to be maintained by each executive for each message it dispatches to the transport layer.

Each message sent from one TWE to another is given a sequence number. When a message is dispatched, a record containing the message's sequence number and timestamp is held by the sending executive. When a message is received from the transport layer, an acknowledgement message is sent back to the event message's source executive containing the sequence number of the latest message received. On receiving the acknowledgement, the source executive knows that the event message has been received and no longer has to be treated as transient. It is now safe to discard the record referring to this event message.

Using this scheme, the source executive can find a lower bound on the timestamps of all the messages it has dispatched that may still be transient. This value, called the *transient message time* (TMT) is the minimum timestamp currently held in a transient message record on this executive.

On each executive the LGVTman can calculate the executive's local minimum time (LMT) from equation (5.1), where EVT is the *executive virtual time.*

$$LMT = min\{\, TMT, EVT \,\} \tag{5.1}$$

The virtual time value sent to the GVTman is called the *reported time* (RT). If a new LMT value calculated is greater than the current RT value (i.e., the last local minimum time value reported to the GVT manager), then the new LMT value is sent to the GVTman and RT set to this value. Note that if the new value of LMT is lower than RT, this must also be reported to prevent the GVTman using the old RT value to erroneously calculate a GVT value lower than this executive's LMT.

In order to assure the correctness of GVT values calculated the algorithm also has to take the simultaneous reporting problem into account. Figure [5-1a], shows the sending of an event assuming that the event message (1) sent from TWE_A to TWE_B has a timestamp lower than RT_B (the RT value TWE_B). On receiving (1) TWE_B sends a new RT message (2) to the GVT

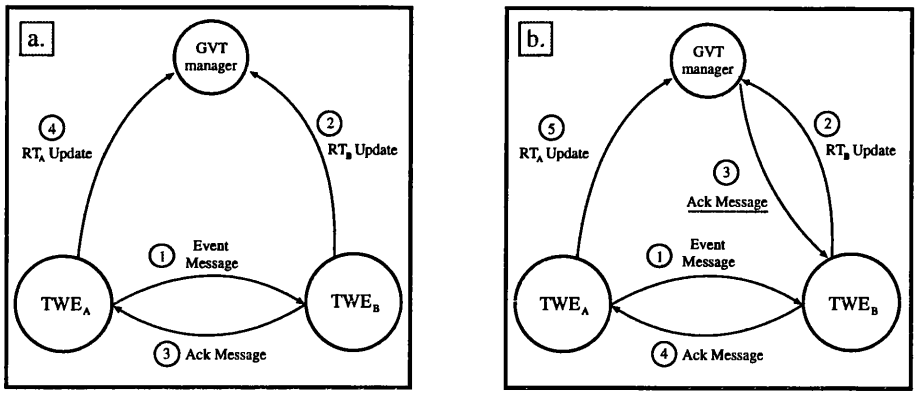

Figure 5-1: Diagrams showing the messages sent between two TWEs and the GVT manager in the situation where the message sent from TWE_A to TWE_B causes LMT_B to decrease. In figure **b** an extra acknowledgement (3) is sent that assures correctness.

manager and sends an acknowledgement to TWE_A (3). Suppose that this increases LMT_A causing TWE_A to send a new RT message (4). As long as the GVT manager reads (2) before reading (4) this series of messages will not result in the GVT manager calculating an incorrect GVT value. It is possible on some communication systems that (4) will be read before (2) creating the possibility that an incorrect GVT value will be calculated before (2) is read. This situation can be prevented by employing acknowledgements for RT messages as shown in figure [5-1b].

In figure [5-1b] the acknowledgement (4) is not sent to TWE_A until the acknowledgement (3) is received by TWE_B. Since it is the timestamp of (1) that caused LMT_B to decrease and this value is included in the calculation of TMT_A and therefore of LMT_A until (4) arrives, using the RT acknowledgement (3) assures the correctness of the GVT calculation.

ALB is similar to the GVT calculation scheme used by the *pGVT* algorithm [15]. The main difference is that the *pGVT* algorithm has a more sophisticated policy for determining when to send new RT values to the GVTman. Different reporting strategies are described in section 5.5.

5.2 Batched Acknowledgements

This section shows how ALB, the simple GVT calculation scheme presented in section 5.1 can be modified to reduce the number of acknowledgement messages required.

In ALB an acknowledgement is sent for each event message received from the transport layer. Although acknowledgement messages are short, most of the time taken to send a message is setup overhead which is independent of the length of the message. Thus, reducing

the number of acknowledgements needed will improve the efficiency of the algorithm. The solution presented here relies on a reliable, FIFO message transport system. I describe the connection between any pair of TWEs as a *channel*, with channel_{AB} carrying message from TWE_A to TWE_B and channel_{BA} carrying messages form TWE_B to TWE_A. Note that this is not the same as a channel in a CMB simulator that links a pair of LPs. In order to work correctly in an unreliable messaging environment, the algorithm would require some modifications. This issue is considered in section 5.7.

In ALB, a sequence number is associated with each message passed to the transport layer. Since the sequence numbers used form a monotonically increasing sequence the acknowledgement of a message with sequence number *sn*, acts as an acknowledgement for all messages with sequence numbers less than or equal to *sn*; this is only true in a FIFO messaging environment. If the acknowledgement can be delayed until several event messages have been received, a single acknowledgement message will act as an acknowledgement for all messages received from the same executive.

If the acknowledgement of a message that arrives on channel_{AB} is delayed until TWE_B sends a message to TWE_A, the sending of the acknowledgement can be avoided completely. This is achieved by appending the sequence number of the last message received on channel_{AB} to each message passed along the channel_{BA}.

To complete this scheme, a policy is required for determining how long to delay sending an acknowledgement. If messages are passed back and forth between executives regularly, this might not be an issue; if this is not the case, delaying the acknowledgement too long will prevent GVT from advancing. The amount to delay sending an acknowledgement is discussed in section 5.5.

5.3 Valley Messages

In the batched acknowledgement scheme presented in section 5.2, records for all messages passed along a channel are maintained until an acknowledgement is received. The number of records that have to be maintained can be reduced by observing a property of the sequence of message timestamps. In a Time Warp system, the timestamps of event messages sent along each channel do not usually form a monotonically non-decreasing sequence. Some messages have a timestamp value that is less than the timestamp of the message that preceded them. A message with this property is described as a *valley message*, Lin and Lazowska [33].

The graph in figure [5-2] shows the timestamps of event messages passed along a channel plotted against the message sequence numbers. The messages with sequence numbers 7, 12, 13, 17, 20, 22 and 23 are valley messages. When a valley message is sent along a channel, the acknowledgement of any greater timestamped messages preceding the valley message will

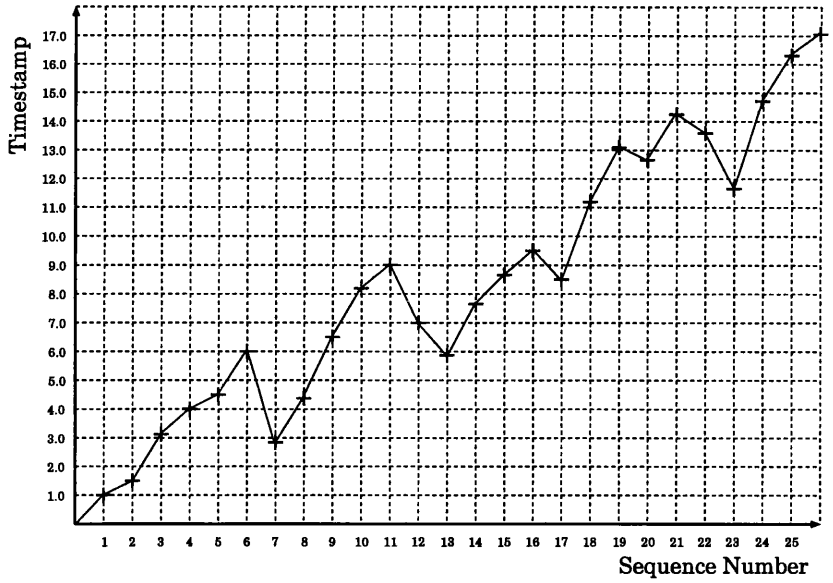

Figure 5-2: Graph showing the relationships between the sequence numbers and timestamps of messages sent along a communication channel.

not increase the channel's transient message time. It should be noted that it is also true that the channel's transient message time will not be increased by acknowledging an equal timestamped preceding message. Although equal timestamped messages were not in Lin's definition of valley messages [33] they can safely be treated in the same way as valley messages in the algorithms presented in this chapter.

Consider the sequence of messages with sequence numbers ranging from 1 to 8 as shown in figure [5-3]. Receiving an acknowledgement for any message in a shaded area of the graph (sequence numbers 3, 4, 5 and 6), will result in a minimum transient message time equal to the timestamp of the valley message that follows. If all the messages before the shaded area have been acknowledged, the minimum transient message time will only be increased by the receipt of an acknowledgement containing a sequence number greater than or equal to that of the valley message.

Now consider the messages with sequence numbers between 18 and 24 shown in figure [5-4]. Message 20 is a valley message that stops the acknowledgement of message 19 increasing our knowledge of the minimum transient message timestamp. Also message 22 is a valley message that removes our interest in the acknowledgement of message 21. Message 23 is another valley message with a timestamp less than the timestamp of valley messages 20 and 22. If none of the messages with sequence numbers between 19 and 22 have been acknowledged by the time that message 23 is sent, receiving an acknowledgement for any of them will not

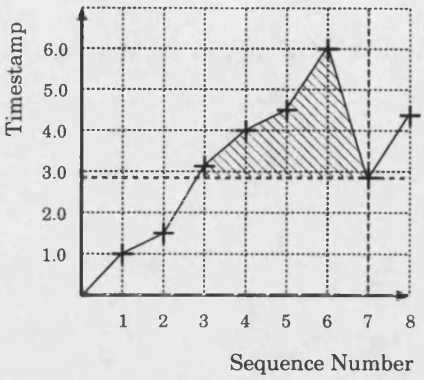

Figure 5-3: Graph showing how valley messages increase the sending executive's knowledge of the transient message time for a channel.

increase the channel's transient message time.

Using valley message information allows some message records to be discarded before an acknowledgement is received. When a message is sent, its timestamp is compared to the timestamp held in the last element of the list of transient message records. If the new message has a timestamp less than the message record's timestamp, the message record can be discarded. This is safe to do since an acknowledgement of the message sequence number contained in the record will not increase the transient message time beyond the timestamp of the message now being sent.

Consider the actions taken by the sending executive when the messages shown in figure [5-4] are sent. The message records maintained on the senders end of the channel are depicted in figure [5-5]. Message 18 is not a valley message so just add a message record to the end of the list. Same for message 19. Message 20 is a valley message so the last message record is removed from the list and the message record for message 20 added. Message 21 is not a valley so a message record is added to the end of the list. Message 22 is a valley message so the message record for message 21 is removed and the message record for message 22 is added. Message 23 is a valley message whose timestamp is less than both message 22 and message 20. Therefore, the records for messages 22 and 20 are removed from the list and the record for message 23 added. Finally, since message 24 is not a valley message, its transient message record is simply added to the end of the list. Thus seven messages have been sent, but only three message records are required to ensure the correctness of the transient message time calculation for this channel. Note that using this scheme the message record list is always sorted in increasing timestamp order even though no sorting is done.

When an acknowledgement message arrives, all message records holding sequence num-

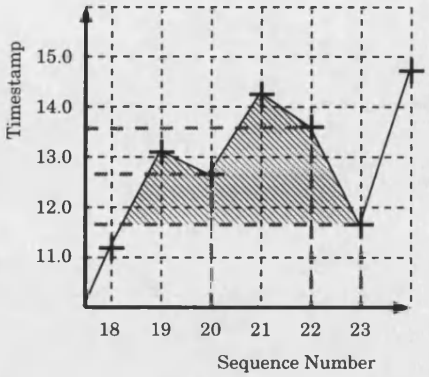

Figure 5-4: Valley message 23 removes the need to acknowledge all messages from 19 through to 22, even though messages 20 and 22 are themselves valley messages.

Seq. Number	18	19	20	21	22	23	24
Message Record List	18:11.2	18:11.2 19:13.1	18:11.2 20:12.7	18:11.2 20:12.7 21:14.3	18:11.2 20:12.7 22:13.7	18:11.2 23:11.7	18:11.2 23:11.7 24:14.7

Figure 5-5: Message records maintained at the sender's end of the channel using the mixed acknowledgement system for the messages shown in figure [5-4].

bers less than or equal to the acknowledged sequence number can be removed from the list. The list is in sequence number order so removing elements is simply a case of removing elements from the front of the list until either the list is empty, or the message record at the front of the list has a sequence number greater than the one contained in the acknowledgement.

Using valley number information in this way achieves two goals. The number of message records that have to be held has been reduced and the need to search the list of message records for the channel's transient message time has been removed. The channel's transient message time is either the timestamp held in the record at the head of the list, or is infinity if the list is empty. This message acknowledgement scheme will be referred to as the *mixed* acknowledgement scheme.

5.4 Valley Only Acknowledgement

In the mixed acknowledgement scheme, only the sequence number of the of the last message received is included in the acknowledgement message. If the acknowledgement message also contains the timestamp of the last message to be received on the channel, it is possible to calculate the transient message time while maintaining even fewer message records at the sending end of the channel. The scheme that will be described now will be referred to as the *valley only* acknowledgement scheme.

Within the sequence of timestamps of messages passed along a channel there will likely be increasing subsequences. In the mixed acknowledgement scheme, message records are added to the message record list for all messages. Some records are later removed as valley messages are sent, but many of the records will have to be maintained until an acknowledgement is received.

In the valley only acknowledgement scheme, only records relating to valley messages are maintained at the senders end of a channel. The valley message records are added to the record list using the same rules used to add elements to the record list in the mixed scheme: i.e., all records containing timestamps greater than the new message record are removed from the end of the list before the new record is added. This results in the list being ordered by increasing timestamp value.

In the valley only scheme it is also necessary to keep the sequence number and timestamp of the last message sent along the channel. The timestamp is required in order to determine if the next message sent is a valley message. The sequence number is required so that it is possible to establish when all messages have been acknowledged.

When an acknowledgement is received, all records containing sequence numbers less than the acknowledged sequence number are removed from the front of the list. In this scheme the timestamp held in the first element of the message record list may not be the transient message time. Instead, if not all messages have been acknowledged, the transient message time is the minimum of the timestamp of the first message record and the timestamp value included in the acknowledgement. If all messages have been acknowledged the transient message time is infinity, just as in the mixed acknowledgement scheme.

This solution is similar to the solution presented by Lin and Lazowska [33]. The main difference between the two algorithms is that Lin's algorithm does not discard any message records until a message acknowledgement is received.

Although this scheme can reduce the number of message records that have to be held at each end of a channel, it can also result in a higher TMT value being calculated than if the mixed acknowledgement scheme is used. To see why this is consider the messages with sequence numbers from 23 to 26 in figure [5-2]. These messages have increasing timestamps

so the sender TWE would have to hold four message records with the mixed scheme, but only the values of the final message's timestamp and sequence number with the valley only scheme. Consider the case where the sender receives the acknowledgement for message 24. With the mixed scheme the LMT is now equal to the timestamp held in the record representing message 25, which is 16.3. With the valley only scheme since no valley messages have been sent in this sequence the LMT is equal to the timestamp value contained in the acknowledgement message, in this case the timestamp of message 24 which is 14.7. So a lower LMT is calculated with the valley only scheme.

5.5 Acknowledgements and Reported Time

This section looks at two closely related problems; when to send message acknowledgements and when to send reported time (RT) messages. As explained in section 5.2, delaying the sending of acknowledgements can cause the number of control messages required to be reduced, but delaying an acknowledgement too long will prevent GVT from advancing. In the ALB algorithm, RT messages are sent whenever a new LMT value is calculated and some RT messages have to be acknowledged.

Two schemes are presented in this section. The first is an asynchronous scheme similar to that used in *pGVT*, D'Souza *et al.* [15]. The second is a round based scheme that eliminates some of the problems that I encountered with the asynchronous reporting scheme.

5.5.1 Asynchronous Reporting

The asynchronous reporting scheme used in ALB (section 5.1) is a simplified version of the reporting scheme used in *pGVT* [15]. *pGVT* uses wall clock time calculations based on timing the wall clock time taken to exchange messages with the GVT manager in order to try to optimise when RT messages are sent. This requires all GVT messages to be acknowledged and wall clock time values to be recorded. Like the scheme used in ALB, if the LMT value at a TWE decreased a new RT message has to be sent in *pGVT*. The authors of *pGVT* did give a suggestion as to how the number of times this happened could be reduced. They suggest sending an RT value lower than the LMT value for executives with EVT values far ahead of the GVT known to the executive. In my experiments it proved very difficult to determine when lowered RT values should be sent; choosing RT values that were too low proved to be very detrimental to the performance of the system when some event execution throttling algorithms (see chapter 6) were also being used.

Another problem I encountered with the asynchronous reporting scheme was that it proved difficult to determine when acknowledgements should be sent when batched acknowledgement was used. Messages needed to be acknowledged in time to allow GVT to advance, but

acknowledging too early increased the number of control messages required.

If either too low RT values were chosen, or event messages were acknowledged too late, performance suffered badly when an execution throttling algorithm was also employed. I never managed to find a method to optimise either of these calculations adequately. These problems eventually led me to adopt the round based reporting scheme described in the following section.

5.5.2 Round Based Reporting

In the round based reporting scheme the local GVT managers have three modes of operation. In **mode 1**, **n** events are executed or a spin loop is used to wait for an equivalent amount of time if less than **n** events are available for execution. After this time, the LGVTman switches to **mode 2** and remains in **mode 2** while a further **m** events are executed. In DTW the values of **n** and **m** are 20 and 10 respectively. Then LGVTman switches to **mode 3**, the LMT calculation is performed and a new RT message dispatched. The LGVTman remains in **mode 3** until a new GVT message arrives at which point it reverts to **mode 1**.

Acknowledgement messages are only sent while the LGVTman is in **mode 2**. When the mode is changed from **mode 1** to **mode 2**, acknowledgements are sent on all input channels with transient message times less than the EVT. Any messages that arrive while the LGVTman is in **mode 2** that have timestamps less than the EVT are acknowledged immediately.

In the round based scheme the GVTman does not calculate a new GVT value until an RT message is received from each TWE. Since message acknowledgement is suspended by each TWE as soon as their RT message is sent, the GVTman cannot calculate a false (i.e., greater than SBT) GVT value. To see why this is consider the following. If TWE_A receives a new message that lowers its LMT after sending its RT message, the new message will not be acknowledged until after the next GVT value is calculated. Therefore, the new message's timestamp will be taken into consideration in the GVT calculation as a transient message at the sending executive. The next GVT value will not be calculated until TWE_A has sent a new RT value that will include information on all active messages held on TWE_A.

Unlike the asynchronous scheme, the round based scheme requires every executive to send an RT message before a new value of GVT can be calculated. This increases the number of messages that will arrive at the GVT manager at one time. The round based scheme does have a major advantage however. Since with this scheme there is no penalty for sending high RT values, there is no reason not to set RT to LMT when a new RT message has to be generated. This removes the possibility of calculating an artificially low GVT value that a windowed RT value in the asynchronous scheme could produce. For this reason the round based scheme is currently the default reporting scheme used in DTW.

5.6 GVT Manager Location

Placing the GVTman on one of the Time Warp executives does have some advantages over placing it on a dedicated executive. With the GVTman on a TWE, the total number of messages needed to calculate and distribute GVT is reduced. The LGVTman on the same executive as the GVTman does not have to send its reported time via the transport layer. The value can simply be passed to the GVTman using a function call. Similarly, the GVTman object can update the value of GVT in the LGVTman using a function call. Apart from this, a separate GVT manager executive needs a PE on which to execute. On a small parallel computer all of the PEs would probably be allocated a TWE, so the GVTman should be placed in a TWE.

There are some disadvantages to sharing an executive however. The GVTman will take up some CPU cycles that could be used for executing events. More importantly, the executive will have to spend far more time reading messages from and writing messages to the transport layer. The greater the number of TWEs, the more messages the GVTman will have to handle, so the less advantage is gained by sharing an executive. For the test simulations for which results are reported in chapter 9 a separate executive was used to run the GVT manager.

5.7 Unreliable Messaging

The GVT algorithms described in this chapter require a reliable FIFO message transport system to operate correctly. On many modern MIMD parallel computers this facility is provided by the computer's communications hardware. On systems where this is not the case, reliable FIFO messaging can be provided in software. Unfortunately, software based reliable message systems can add a considerable overhead to the cost of sending each message.

Like many modern message passing systems, DTW uses the MPI message passing library, which guarantees reliable messaging. It would be possible to modify the algorithms presented in this chapter for use in an unreliable message passing environment using techniques similar to the ones discussed by Lin and Lazowska in [33].

5.8 Other Algorithms

Since completing this work another GVT algorithm for distributed memory environments has come to my attention. Several ideas for GVT algorithms were presented by Mattern in [41]. The most interesting of these uses different phases of operation to indicate the GVT calculation status of a particular executive. It also marks each message with the GVT calculation state of the sending process. There are two states described as *white* and *red*. When an executive is not in a GVT calculation phase it is in the white state and switched to the red state when the

GVT calculation phase begins. On switching to the red state an executive saves the current value of its EVT. Also, each executive monitors any incoming white messages since these will not have been taken into account in the senders LMT calculation. Therefore the minimum timestamp value of any white messages received is maintained by the executive. Using the recorded EVT values and the minimum value of the white message timestamps a new GVT value can be calculated, but only once all of the transient messages are known to have been received. Mattern described several methods for determining when all the transient messages have arrived, with the most interesting being a scheme that uses vector counts of all messages sent to and received from each executive. By summing the vectors from each executive it is possible to determine if all the messages sent in the last white phase have been received. In the scheme presented each executive reports their EVT value, minimum white message timestamp and vector counts to a central GVT manager. Several rounds of messaging could be required since the new GVT value is not found until the sums of all the vector counts are zero. Using these counters means that no message acknowledgements are required and FIFO messaging is not essential.

An extended version of Mattern's vector clock based algorithm is presented by Fujimoto and Hoare in [21]. In this a parallel reduction algorithm is used to calculate the minimum EVT value. This allows a new GVT value to be calculated in log N time on N executives. In this each executive performs log N pairwise minimum operations before a new global minimum time value is received. The algorithm checks for transient messages by keeping counts of the number of messages sent and the number received and adding the difference of these values to the value held in the message that is being used to calculate the local minimum. After a single round of messaging is likely that this message count will not be zero, so a second reduction operation is initiated. After the second reduction operation all of the transient messages will have been accounted for. The use of counters means that like Mattern's algorithm, no message acknowledgements are required and FIFO messaging is not required. Another advantage of this scheme is that it does not require a centralized GVT manager to perform the GVT calculation, since all the executives receive the new GVT value as part of the reduction algorithm. It does require slightly more messages to perform the calculation than the algorithm described in this chapter, but these are distributed evenly through the system which could eliminate the possible point of contention at the centralized GVT manager.

I have not had time to assess the relative performance of the algorithm presented in this chapter, and the one presented in [21]. The algorithm presented in [21] clearly has some advantages and it could be that combining features of both algorithms could be advantageous. This is left as further work.

5.9 Summary

This chapter has considered the problems associated with calculating the global virtual time in a distributed programming environment. A GVT calculation algorithm suitable for using with the event scheduling scheme presented in chapter 7 has been developed and implemented in DTW.

Two methods were considered for reducing the number of message acknowledgements required, the mixed acknowledgement scheme and the valley only acknowledgement scheme. Each of the schemes was shown to have some advantages and some disadvantages over the other. Currently the mixed acknowledgement scheme is employed in DTW; the cost of possibly calculating higher transient message times with the valley only scheme outweigh the slight additional storage costs with the mixed acknowledgement scheme.

Chapter 6

Event Scheduling

A major part of this thesis work has involved the development of an event scheduling algorithm that will be introduced in chapter 7. This chapter considers the issues related to event scheduling and reviews previous work in this area. The previous algorithms are only described in outline. Due to the different ways in which a Time Warp system can be constructed, in particular the differences that result from using a single-LP or multi-LP model, explaining the previous algorithms in great detail is likely to lead to confusion.

This chapter is laid out as follows. Section 6.1 describes the scheduling problem, how the construction of the Time Warp system can effect which algorithms can be implemented and how a particular algorithm can cause the system to behave. Then, section 6.2 looks at some of the scheduling schemes tried previously; you will see from this section that this is an area that has received a great deal of attention. The chapter is summarised in section 6.3.

6.1 The Scheduling Problem

This section describes what is meant by scheduling and throttling, the differences between scheduling events and scheduling LPs and the types of algorithms used to perform these tasks. It goes on to explain some of the problems associated with throttling and the different ways in which throttling can be applied.

6.1.1 Scheduling and Throttling

The algorithms that will be discussed in section 6.2 of this chapter have been presented to solve two separate problems in Time Warp. The two problems in question are event scheduling and execution throttling. The distinction between these two categories being that event scheduling algorithms pick the next event to execute, while throttling algorithms make a decision regarding if a particular event should be executed at the current wall clock time. Therefore an event could

be scheduled for execution by a scheduling algorithm but the execution of the event could be delayed by a throttling algorithm. The idea of execution throttling is to attempt to prevent too many events from being executed early; this in turn can reduce the occurrence of the so called "failure modes" described in section 4.8.

The algorithms described in Choi *et al.* [9], Ayani *et al.* [1], and by Som and Sargent [56], are presented as event scheduling algorithms. Most of the other algorithms discussed are presented as execution throttling algorithms. Note that although [56] is presented as an event scheduling algorithm it is very similar to the algorithm presented by Ferscha *et al.* [17] which is presented as an execution throttling algorithm. The chief difference in these two algorithms is that the algorithm described in [56] uses a multi-LP model that allows the execution of events and blocking of LPs to be interleaved on the same TWE, while the algorithm in [17] uses a single-LP model resulting in either execution or blocking. On the other hand, the penalty based execution throttling scheme described by Reiher *et al.* in [46] also interleaves event execution and LP blocking using a multi-LP model and could easily be described in terms of event scheduling; i.e., it schedules events believed to be more likely to be correct.

It would be possible to view many of the throttling algorithms as event scheduling algorithms in this way if they were implemented in a system using a multi-LP model in which the throttling decision is made individually at each LP and a blocked LP (i.e., an LP that the throttling algorithm has determined should not execute its next event at this time) relinquishes the thread of execution to non-blocked LPs. While it is usually clear if the author intended the algorithm to be executive wide or per LP in a multi-LP model this is often stated as part of the implementation rather than as part of the algorithm. For this reason all of these execution throttling algorithms could also be classed as event scheduling algorithms.

6.1.2 Scheduling Events or LPs

Some systems, including DTW, act by scheduling LPs to execute events rather than scheduling events directly. I group LP scheduling algorithms in the same class as event scheduling algorithms since for the most part the two methods are equivalent. If an LTF scheduler (see section 2.2.2) is used, the semantics of scheduling the LP with the lowest timestamped event to execute the event, or scheduling the lowest timestamped event and then executing it at its destination LP are equivalent if in the LP scheduling case, the LP is returned to the scheduling queue after executing a single event.

The only real differences are in the efficiency of the implementations of different scheduling policies with the different scheduling schemes. With LTF scheduling it is arguable that a direct event scheduling scheme would be more efficient than an LP scheduling scheme, since it is likely that the event can be scheduled using less instructions than if first the LP is scheduled and then the event is removed from the LP's event queue. The LP scheduling scheme

has the advantage that the event queues themselves are short so the actual advantage of event scheduling over LP scheduling when LTF scheduling is used will depend greatly on the implementation of the queues and how well the queues perform given the sequence of events that occur in a particular simulation.

Where the LP scheduling method can come into its own is where it appears best to execute an event at an LP other than the LP with the lowest timestamped event. With an event scheduler using a single event queue per executive, a search may have to be performed for the next event for an LP, while if an LP scheduler is used with per-LP event queues, whichever LP is scheduled the next event for that LP will be directly accessible. Also, using per-LP event queues allows more than one event to be executed at an LP at one time without having to search for the next event. It will be shown in section 9.2.1 that doing this can improve the performance of the system.

6.1.3 Algorithm Types

The throttling algorithms work by inhibiting event execution when it is determined that there is a low likelihood of the next event for an LP being the next event that should be executed by the LP. Some algorithms use constant parameters to throttle event execution. Although such algorithms can be extremely effective in some cases, Turner *et al.* [69], the selection of inappropriate throttling parameters has been shown to be highly detrimental to performance, Reiher *et al.* [46]. Due to this, other algorithms dynamically adjust the values of the throttling variables to optimise the throttling adaptively.

The so called *adaptive algorithms* use some heuristic to determine if too much or too little throttling is being applied at any time. Some use statistical information based on the arrival of events while others use statistical information on the occurrence of rollbacks or on the generation of anti-messages. A few use global information based on how an executive's EVT is progressing with respect to the EVT of other executives. Most use only virtual time in the calculation, but a few, notably those presented by Ball and Hoyt [3], Hamnes and Tripathi [28], and by Ferscha *et al.* [17][18], use a combination of wall clock time and virtual time values.

The throttle applied by these algorithms is either in the form of a virtual time block or wall clock time delay. A virtual time block suspends execution until the difference between the EVT and GVT decreases; this occurs when either GVT increases or when a new lower timestamped event arrives at the executive. A wall clock time delay based block suspends execution for some length of wall clock time. Some schemes applying wall clock blocks resume execution as soon as a lower timestamped event arrives, while others remain blocked until the calculated blocking time is complete.

Another group of algorithms are described by Ferscha and Luthi in [18] as implicit throttling schemes. These control some parameter, usually the amount of memory used, that has

a throttling effect on execution and can reduce thrashing behaviour even though they are not directly controlling when events are executed.

6.1.4 Problems with Throttling

While throttling can reduce the thrashing behaviour observed when the system enters one of the failure modes explained in section 4.8, it can also result in lost opportunity. The fact that optimistic simulators can outperform conservative simulators for some classes of simulations is because they are able to take advantage of situations where interactions could take place, but in reality, they do not. Inappropriate throttling will remove this advantage. A simple cost model for throttling given by Srinivasan and Reynolds in [59], is shown in figure [6-1]. This shows three overlaid graphs plotting costs against the amount of throttling (restriction of optimism) applied. Plot (**i**) shows the costs associated with over-optimistic event processing and plot (**ii**) shows the cost of lost optimism. Plot (**iii**) shows the overall cost derived by adding the values used for plots (**i**) and (**ii**). The balance point marks the minimum cost point, marking the degree of restriction of optimism that should yield the best performance from a simulation.

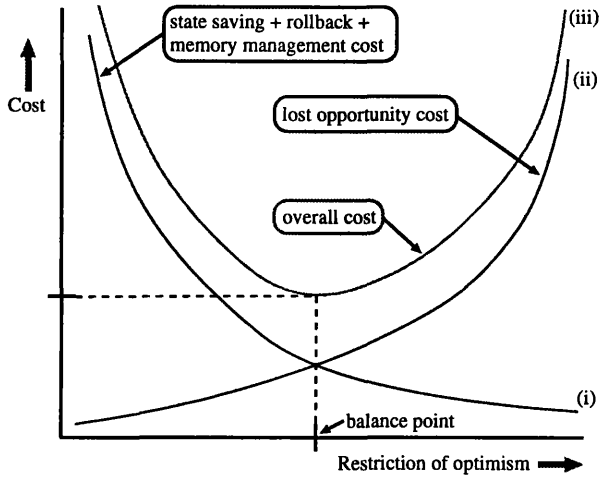

Figure 6-1: Diagram taken from Srinivasan and Reynolds [60], that plots the costs involved in over-optimistic event processing and in losing opportunity due to excessive restriction of optimism. Plot (**iii**) shows the overall cost, with the balance point representing the optimal blocking behaviour.

The costs shown in figure [6-1] provide a useful way of thinking about balancing event execution with blocking. Runtime minimisation of the the function described by plot (**iii**) is not easy, since the cost of restricting optimism, plot (**ii**), is difficult to determine dynamically. Also, while it is possible to minimise the function statically given a complete simulation run, the

optimal amount of optimism restriction is likely to change during the course of the simulation.

The term *asynchronous adaptive waiting protocols* (AAWPs) was introduced by Srinivasan and Reynolds in [59]. An AAWP is a throttling algorithm where the determination of the throttling parameter is performed locally to each executive. For example, if an algorithm uses a global window in which events can be executed, the algorithm is not an AAWP. On the other hand, if the size of the window is determined individually by each executive, then that algorithm is an AAWP. An example is given in [59] where a faulty blocking decision, i.e., where it is decided to block when it was not necessary to do so, increases the number of rollbacks by delaying the generation of events that are required by other LPs. This is a general problem of AAWPs.

6.1.5 Risk and Aggressiveness

The concepts of risk and aggressiveness were explained in section 1.1.1. All but one of the throttling algorithms presented in section 6.2 throttle event execution by limiting aggressiveness. Limiting aggressiveness also limits risk, since if events are not executed far ahead of GVT, no event will be generated with a VST far ahead of GVT and therefore risk is reduced. Another form of throttling is to simply limit risk by preventing messages from being despatched until the GVT gets close to the message's VST. One algorithm in section 6.2 works by limiting risk.

The term *Aggressive No Risk* (ANR) has been used to describe algorithms that work by completely removing any risk. An ANR simulator has the advantage that no false event messages are ever exchanged between LPs. A comparison between Time Warp and an ANR system was presented by Bellenot in [4]. This showed that only simulations with the property that message lifetimes were long in comparison the average LVT increase per event execution gave good performance in the ANR system. Other ANR algorithms are presented by Reynolds in [48] and by Steinman in [66].

While I believe it is useful to understand the differences in the costs associated with risk and aggressiveness, I believe the primary role of an execution throttling scheme must be to control aggressiveness. With uncontrolled aggressiveness, many events can be executed early leading to long rollbacks. Just limiting risk could increase the number of rollbacks due to delaying the arrival of events at their destination LPs. One area where controlling risk could be beneficial is for events with long lifetimes. In this case removing all the risk from the dispatch of the event will be less likely to cause a rollback than if the same is done for an event with a short lifetime. This issue will not be pursued further here.

6.2 Previous Algorithms

This section outlines some of the event scheduling and execution throttling algorithms that have appeared in the literature. First, direct throttling algorithms are presented, with static algorithms in section 6.2.1 and the adaptive algorithms in section 6.2.2. Then some implicit scheduling schemes are presented briefly in section 6.2.3.

6.2.1 Static Algorithms

Static throttling algorithms have been presented by Reiher *et al.* in [46], by Turner and Xu in [69] and by by Choi and Min in [9]. A *window-based throttling* scheme is presented in [46]. With this only events whose VRTs fit into a fixed sized window above GVT are executed. This ensures that the LVT of every LP in the entire system lies within this window. Though some success was reported in some instances, very marked slow downs were reported where the window size selected was too small. Making the window too large would allow thrashing behaviour to occur. The need for user selection of the window size and the poor performance resulting in bad parameter setting led to the conclusion that such a scheme was not good for a general purpose simulator. Note that there are some similarities between *window-based throttling* [46] and the often cited *moving time window* (MTW) algorithm presented by Sokol and Stuckly [54]. MTW does not ensure events are executed obeying the local causality constraint and as such has little relevance to this thesis work.

A slightly different approach to execution throttling is taken by Turner and Xu in the *Bounded Time Warp* (BTW) algorithm [69]. Unlike window-based throttling [46], BTW executes all of the events in the current virtual time window and performs a global synchronisation before moving to the next time window. This has the effect of ensuring that any false events generated in one execution window will have been cancelled and removed from the system before work starts in the following execution window. Notice that this prevents vortices (see section 4.8) from extending past the next virtual time window. Good results are reported from an example telecommunications simulation, though as with window-based throttling, selecting a small window size resulted in poor performance.

A different approach to throttling was taken in the *penalty-based throttling* scheme also presented by Reiher *et al.* in [46]. In this, LPs are penalised when they generate anti-messages, the idea being to give less CPU time to LPs that have recently been behaving in an overly optimistic manner. The penalty takes the form of the LP being skipped over by the LP scheduler for a number of times directly proportional to the number of anti-messages that the LP has sent. The value of the constant of proportionality used for determining how many times a penalised LP has to be skipped for each anti-message sent has to be set by the user. This system is interesting since it uses a multi-LP model and interleaves event execution with LP blocking.

However, the reported performance using this scheme is disappointing. The paper does point out that there are other metrics that could be used to determine when an LP should be penalised, and maybe the use of anti-messages generated was inappropriate.

A pair of algorithms are described by Choi and Min in [9]. Both work by executing events in batches, with interprocessor communications only being performed when a bound is reached. The *Balanced Progress by Execution Chance* (BPEC) algorithm works using event counts. A count is kept of how many events are executed by each LP. If an LP has executed the statically defined maximum number of events since the last inter-processor communication, it is passed over for the rest of the executives current execution session. Once the total number of events reaches a statically defined maximum, the current execution session ends with a period of inter-processor communication. The *Balanced Progress by Virtual Time Window* (BPVTW) algorithm simply executes all the events within a statically defined virtual time window before performing interprocessor communications. Both of these algorithms appear to run the risk of failing to dispatch event messages soon enough to avoid causing rollbacks at the receiving LPs.

6.2.2 Adaptive Algorithms

The *filtered rollback algorithm*, Lubachevsky *et al.* [36] is an extension to the conservative *bounded lag algorithm*, Lubachevsky [38]. In the bounded lag algorithm a search is performed amongst adjacent LPs to find a safe time up to which events at a particular LP can be executed. In filtered rollback, instead of computing an absolute bound on the safe time for the current event execution session, an approximation is used. This is done by using half the mean inter-arrival time on each channel in the calculation, rather than using the minimum lookahead on each channel[1]. It is not made entirely clear from the paper if the authors intended for these mean values to be provided by the user, which would make this a static algorithm, or for them to be calculated by the system. I place it in the adaptive algorithm section since dynamic calculation of this value is possible.

In the *Adaptive Time-Warp* (ATW) algorithm by Ball and Hoyt [3], wall clock time delays are used to slow the progress of an executive. This delay is imposed before the execution of an event. The length of the delay is calculated by solving numerically a function that includes the amount of wall clock time spent rolling back and the rate of change of the proportion of time spent rolling back as the delay value is adjusted.

The *Breathing Time Warp* algorithm, Steinman [65] is based on an ANR simulation algorithm called *Breathing Time Buckets*, Steinman [62]. Breathing Time Buckets works by executing all events with VRTs less than or equal to the lowest VRT of any event generated on the executive in the current execution cycle. The executives then synchronise and find the minimum local bound time; i.e., the lowest VRT of any event generated in the whole system in this

[1]The bounded lag algorithm does not use channels, but does use a spatial notion that is analogous to channels.

execution cycle. This minimum bound time then becomes the global bound time and all events with VRTs less than this are committed and all messages generated them are dispatched to their destination LPs. Since this algorithm is risk free, no anti-messages are required. Breathing Time Warp builds on this by starting an execution session with a Time Warp phase (i.e., events are sent with risk). After a fixed number of events have been executed, the execution mode changes and the system uses the rules laid down by the Breathing Time Buckets algorithm. Note that a message might arrive with a VRT such that if it had arrived earlier, it would have been one of the events executed in the Time Warp phase. In this case the system reverts to Time Warp execution (i.e., sending any resulting events with risk) to execute this event. Otherwise, the event execution continues until a local bound time is found using the Breathing Time Bucket rules. At this point the executives enter a synchronous GVT calculation phase that ends with a new GVT value being dispatched to all executives which triggers the next execution cycle starting with a Time Warp phase.

The *Aggressive Adaptive-Risk* algorithm, Soliman and Elmaghraby [55], operates by adaptively adjusting the amount of risk applied to message sending. It does not directly address the aggressiveness of event execution. Event messages for LPs on other executives are buffered and only dispatched at wall clock time valued intervals. During each release only event messages whose timestamps fit into an adaptively sized virtual time window are released. The size of the window is adjusted to try to minimise the number of false events sent while also not delaying sending events that would then arrive at their destinations as stragglers. Note that this algorithm uses the message timestamp (VRT) to determine when the message should be dispatched. As explained in section 4.5, the message's VST would be more appropriate since it is the VST that determines if a message is correct or not. Also, the method used requires the list of buffered messages to be sorted into increasing VRT order. The VST values of messages will always fall into the range [GVT, LVT] and will often be generated in increasing order. Thus the use of VSTs could also reduce the amount of sorting required.

The *local adaptive protocol*, Hamnes and Tripathi [28], uses channels and null messages. It uses a wall clock time delay to suspend execution, though blocking is terminated if an a event with a lower VRT arrives. The calculation of the delay uses a combination of virtual event inter-arrival times and recorded wall clock times of event arrivals. The statistics are calculated for each channel, though only mean values are maintained.

The *self-adaptive* algorithm, Ferscha and Chiola [16], uses forecasting to try to determine the timestamp of the next event. It then makes a probabilistic decision based on the difference between the forecast next event time and the observed next event's VRT. If the next event's VRT is greater than was forecast, a wall clock time delay approximately equal to the wall clock time taken to execute one event is imposed. This algorithm uses null messages and relies on static knowledge of the distribution of event inter-arrival times.

The *Probabilistic Adaptive Direct Optimism Control* (PADOC) algorithm, Ferscha [17], again uses forecasting to determine the probability that an event waiting to be executed is really the next event that should be executed. Although channels are mentioned in the paper the algorithm does not rely on them for correct operation. Again, event execution is suspended for the wall clock time taken to execute one event when the probability of the next event being correctly ordered is low. In this algorithm, a new test is performed after the delay expires thus allowing the actual delay time to increase incrementally until the event probability decision function succeeds. Several schemes are presented for forecasting event arrives. These include arithmetic mean, an exponentially smoothed average and an approximation to the median. Finally, a scheme using *auto regressive integrated moving average* (ARIMA) modelling is presented that overcomes some problems the simpler schemes have in forecasting event arrivals when the inter-arrival times are correlated. The use of ARIMA modelling in this context is expanded on in a follow up paper by Ferscha and Richter [19].

The probabilistic event scheduling scheme presented by Som and Sargent in [56] again operates by forecasting event arrivals based on inter-arrival times. In this sense it is similar to PADOC [17]. A major difference in this scheme is that it is designed for use with a multi-LP model and like the penalty-based algorithm [46] allows event execution and LP blocking to occur concurrently on the same executive. This scheme uses a histogram to define the probability density function modelling the probability of a particular inter-arrival time occurring in the event stream to a particular LP. The selection of which event to execute next is made purely using the probability of the event being correctly ordered; the event with the highest probability of being correctly ordered is executed.

An optimism control algorithm based on minimising rollback costs is presented by Ferscha and Luthi in [18]. This uses a cost model based on wall clock time calculations. Before an event is executed the estimated cost of executing the event and then rolling back is compared to the estimated cost of blocking. If the cost of blocking is lower a further calculation is performed to determine the optimum wall clock time delay to impose.

In the *Parameterised Time Warp* (PTW) system, Palaniswamy [42] and, Palaniswamy and Wilsey [44], two levels of scheduling are employed. The throttling algorithm, *Bounded Time Windows*, is similar to *window-based throttling*, Reiher *et al.* [46], but employs an adaptively changing window size. The window size is adjusted by comparing how much useful work has occurred in the current GVT round compared to previous GVT rounds. The measure, *useful work*, is calculated as a function of counts of events executed, events rolled back, events committed, and anti-messages sent in the calculation period. A separate LP scheduler schedules LPs based on the amount of useful work the LP is expected to do. The useful work value is calculated each time a new GVT value is received. LPs are then scheduled in the order of useful work to do, but revert to round robin scheduling once each LP has had one execution

session in the current GVT round.

The *Elastic Time Algorithm* (ETA), Srinivasan and Reynolds [60], uses what the authors describe as *Near Perfect State Information* (NPSI). NPSI is global information that is updated very frequently. In the case of the ETA, the NPSI comes in the form of GVT values calculated using a high speed reduction network, Srinivasan and Reynolds [58]. While this does limit the usefulness of ETA on clusters of workstations with conventional interconnection networks, some high performance interconnects such as Myricom's Myrinet can achieve performance that approaches that of dedicated reduction networks, Srinivasan *et al.* [57]. ETA works by imposing a real time delay based on how the current *error potential* (EP) compares with the maximum error potential (MaxEP) observed so far. Here the error potential is the difference between GVT and the next event's timestamp. The delay value is continually updated while in the delay loop to take account of increases in GVT and possible new event arrivals. The algorithm requires a user specified scaling value to determine how long the delay should be given the relative values of EP and MaxEP.

A cost model for throttled execution is developed by Das in [11]. This estimates the roll-back probability based on past behaviour. The model developed does not rely on real time delays. Instead it blocks based purely on the difference between the next event's timestamp and GVT. This relies on continuous GVT update making it more suitable for shared memory multiprocessors than for distributed memory computers.

The *adaptive throttle scheme*, Tay *et al.* [68], adjusts the number of events that are executed without checking for external messages. This number can be zero to stop any events being executed before checking for new messages. The number of events to execute is adjusted depending on an LP's progress with respect to other LPs in the system; note that a single-LP model is assumed. The adjustment is made by comparing the LVT of an LP with the LVT values of other LPs. This is achieved by collecting the LVT of LPs along with RT values and distributing the maximum and minimum reported LVT values along with new GVT values. An LP then increases or decreases the number of events executed between checking for external events depending on if its LVT is closer to the minimum or maximum LVT values received from the GVT manager. This scheme seems to ignore the fact that executives with low EVTs may be regularly exchanging messages with other executives with low EVTs. In this case the LPs involved will continually rollback many of the events executed between reading new messages since the event that causes the rollback will not be read in time to prevent it from being a straggler,

The *Minimum Average Cost* (MAC) *Adaptive Synchronisation* algorithm, Mascarenhas *et al.* [40], is a channel based algorithm described by the authors as being based in part, on ideas presented by Ball and Hoyt in [3]. With its channel based architecture it shares much with the local adaptive protocol, Hamnes and Tripathi [28], though the statistics collection and

blocking decision functions used in MAC are more complex. Inter-arrival distributions are estimated in terms of both wall clock time and virtual time, and used along with other cost functions to estimate the rollback probability. A real time delay is imposed if this probability is high, with the delay time determined by substituting the probability found into a cost function.

6.2.3 Indirect Throttling Schemes

The algorithms in the two previous sections all worked by determining when an event should be executed based on the timestamp of the event. Another way to throttle event execution is to place a limit on the amount of memory available to the executive. This is based on the principle that thrashing executives will tend to use more memory than well behaved executives, Das [12].

The *Adaptive Memory Management* (AMM) protocol, Das and Fujimoto [13], uses *cancelback*, Jefferson [30], with dynamically adjusted parameters to adaptively change the amount of memory available to each executive and therefore adjust the ability of LPs to generate new events and advance their LVTs. The parameter adjustment is performed based on changes in the ratio of events committed to events rolled back between calls to the cancelback procedure. This is called at fossil collection if not before. The implementation of AMM requires a global pool of memory buffers making it unsuitable for use on a distributed memory multi-processor.

The *adaptive flow control* algorithm, Panesar and Fujimoto [45] is another algorithm that adjusts the number of event buffers available to throttle execution. Again, this algorithm is designed for use on shared-memory multi-processors, but unlike AMM, it should be possible to use it on a distributed memory machine with a high speed interconnection network. It works by finding the number of buffers that are needed to hold the events sent from LP_i to LP_j at any instant of time. This number is then increased to take account of the maximum difference between the SBT and the GVT that might be observed. Finally the number is increased to allow for optimistic execution. The size of this final increment is determined by whether or not the rate of event commitment has improved since the last change was made. The size of the final increment starts off small and increases with each new calculation until performance starts to degrade.

6.3 Summary

This chapter introduced the problems of event scheduling and outlined some of the event scheduling and execution throttling algorithms developed to attempt to overcome these problems. The event scheduling and execution throttling algorithms have been explained together since they have a great deal in common. Often the difference between whether an algorithm is classified as being one or the other is determined by whether any throttling decision is made on a per-LP basis or a per-executive basis.

Many event scheduling policies have been proposed, but none can really claim to have completely solved the event scheduling problem. It is known that static event scheduling algorithms can cause poor performance if inappropriate throttling parameters are used for a particular simulation. It is also known that adaptive algorithms can lead to poor performance particularly if they cause event execution to be throttled for correctly ordered events while allowing out of order events to be executed. This situation can occur if the algorithm fails to adapt fast enough to changes in the workload in different parts of the model. In the following chapter a new LP scheduling algorithm is introduced that uses many ideas from the algorithms presented in this chapter and attempts to overcome some of the problems that can lead to poor performance with these algorithms.

Chapter 7

Belief Based Scheduling

This chapter describes the scheduling algorithm that I have developed for use in DTW. The algorithm, known as *belief based scheduling* (BBS), acts by scheduling LPs to execute events rather than scheduling events directly. Once an LP is scheduled, a separate decision process is used to determine which events to execute. This makes the scheme similar to that used by CMB simulators (see section 1.1.2) except that the LP local event execution decision is based on a belief that each event is correctly ordered, rather than the absolute certainty afforded by the channel information in a CMB simulator.

This chapter is laid out as follows. First the aims of the scheduling algorithm being developed are presented in section 7.1. Then the algorithm is outlined in section 7.2. The next four sections give details on the theory behind the algorithm and the chapter is summarised in section 7.7.

7.1 Scheduler Aims

The main aim in producing this algorithm was to produce a scheduling algorithm that would give acceptable performance from the simulator without the need for the simulation modeller needing to spend time tuning system parameters. The aim of getting "acceptable performance" may seem somewhat conservative, but in reality Time Warp systems usually require a considerable amount of tuning to achieve good speedups. To be able to achieve a high proportion of the maximum speedup available without such tuning is desirable in many cases.

In order to achieve this goal, it is clear that the algorithm needs to have some method of limiting the onset of the so called "failure modes" described in section 4.8. Also, the algorithm has to be able to adapt to the behaviour of a particular simulation, not only so that it can handle new simulations without tuning, but also so that it can modify its behaviour to handle changes in the work load that occur during the course of the simulation of a dynamic system.

Other guidelines that were followed were to attempt to take advantage of cache locality if possible and to avoid excessive computational overhead. It was not clear how easy these guidelines would be to follow.

7.2 Algorithm Outline

This section outlines the ideas used in the *belief based scheduling* (BBS) algorithm, with details of the theory used appearing in later sections. BBS uses forecasting at each LP to determine if the next event waiting to be executed at the LP is really the next event that should be executed at the LP. This is not a new idea; there are several algorithms described in chapter 6 that use forecasting in a similar way, though most use forecasting on an executive wide scale rather than at the level of individual LPs. The important features of BBS are:

- The algorithm provides a way of interleaving blocking and event execution at different LPs on the same executive.
- The algorithm promotes the use of what I describe as *multi-event execution* (MEE). In this an LP attempts to execute more than one event each time it is scheduled. The aim of MEE is to improve cache locality and to reduce the overall scheduling cost by amortising the cost of scheduling each LP over all the events executed in the LP's execution session.

Experimentation with forecasting the arrival of events at LPs made it clear to me that this is in general an imprecise science. While the sequences of timestamps given to events generated at any LP are usually deterministically generated by pseudo random number sequence generators, this does not in general give much information on the sequence of timestamps of events that are executed by any LP. This realization led me away from the use of complex statistical techniques for the forecasting; my belief is that in general it is not possible to understand the arrival distributions well enough to make the use of complex statistical techniques effective. Added to this, the complexity of the algorithms used to implement sophisticated techniques is greater. Therefore the cost of computing the forecasting calculation with these algorithms is greater. This means that their use could only really be justified if the quality of the forecasting was greatly increased, but since in general the event arrival sequence at any LP will not be well understood, any large increase in quality is unlikely.

Due to the realization that arrival sequences will rarely be fully understood, BBS uses statistical information and probability calculations only as a component of the heuristics used to determine when an event should be executed. This is the reason that the word "belief" appears in the title of the algorithm rather than "probability". While probabilistic calculations are used, there are too many unknowns in the calculations to say that the result of any calculation is the true probability of an event being correctly ordered.

The aim is to find a value for the belief that the next event observed at an LP is the next event that should be executed at the LP. A value called the *execution belief level* (EBL) is used to determine if an event should be executed. If the belief in an event is greater than or equal to the EBL the event is executed, otherwise the event is not executed and the current execution session for this LP is complete. To achieve MEE, events can be executed at the scheduled LP until an event is encountered for which the belief value fails to reach the EBL. In fact the number of events executed in any execution session is limited by another variable; the reasons for this are explained in section 7.4.

7.3 Statistics and Probability

As was done in the system presented by Som and Sargent [56], BBS uses probability calculations to determine if a particular event is likely to be the next event to be executed by an LP. These calculations are performed based on the stream of events seen at a single LP, where the LP's stream of events is the ordered set of events that have arrived at the LP so far.

The system considers the probability that the current event being considered, which will be referred to as the *candidate event*, is really the next event that should be executed by this LP. Note that any event with a VRT greater than GVT may be rolled back by the arrival of an earlier timestamped event. The system is trying to determine the probability that the candidate event will not later be rolled back.

7.3.1 Probability Distributions

A *probability density function* (PDF) defines the probability that a random variable from a particular distribution takes a particular value. The value of the PDF at x, denoted $f(x)$, represents the probability that a random variable from the distribution will take the value x, i.e., $f(x) = P(X = x)$.

The aim is to determine the likelihood that a candidate event is the next event that should be executed by an LP. This problem could be treated as a counting problem. An interval of virtual time could be selected and the number of events occurring in that interval taken as the random variable. A PDF could be constructed that provides the probability that a particular number of events occurs in an interval of this size. The likelihood of the candidate event could then be determined by considering the number of events in the interval ending at the candidate event's timestamp. While this scheme is valid, an efficient implementation would be difficult due to the need to keep updating where the current interval under consideration begins.

Instead, the PDF used by BBS describes the occurrence of particular *inter-arrival times* (IATs) at an LP. An IAT is the difference between the VRTs of two events executed at the same LP. Figure [7-1] shows a stream of events, $e_1, e_2, ..., e_7$, at an LP with the corresponding IATs,

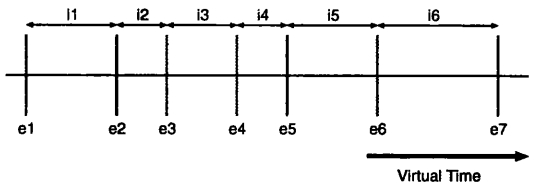

Figure 7-1: Stream of events for one LP with the IAT values shown above.

$i_1, i_2, ..., i_6$. Note that the IAT is not the same as the lifetime of a message since many events could be executed at an LP in the interval of simulation time spanned by the lifetime of any one of the events executed. The value of $f(x)$ gives the probability that an IAT will take the value x. This allows the probability that a particular IAT will be observed to be determined.

The probability required by BBS is the probability that there should be no more events between the candidate event and the event occurring previously in the stream. This is equivalent to the probability that the IAT between any two events will be greater than or equal to the observed IAT, i.e., $P(X \geq x)$ where x in the observed IAT.

The value of $P(X \leq x)$ can be found by considering the value of the *cumulative distribution function* (CDF) at x. The CDF, denoted $F(X)$, is calculated from equation (7.1) if the PDF is continuous and from equation (7.2) if the PDF takes discrete values.

$$F(x) = \int_{-\infty}^{x} f(w)dw \qquad (7.1) \qquad\qquad F(x) = \sum_{w \leq x} f(w) \qquad (7.2)$$

The value of $P(X \geq x)$ can be found using (7.3), but only the value of $F(x) = P(X \leq x)$ is known.

$$P(X \geq x) = 1 - P(X < x) \qquad (7.3)$$

The value of $P(X < x)$ can be approximated using equation (7.4) where ϵ represents the smallest non-zero difference between any two non-equal IATs that have been observed.

$$P(X \geq x) = 1 - P(X < x) \approx 1 - P(X \leq x - \epsilon) = 1 - F(x - \epsilon) \qquad (7.4)$$

The quality of this approximation will depend on the value of ϵ and the way in which $F(x)$ is represented. The representation used for $F(x)$ in DTW is discussed in section 8.1.

7.3.2 Time Effects

The value of $P(X \geq x)$ gives the probability that no event will occur in the virtual time interval $(0.0, x]$. Consider the situation shown in figure [7-2]. In this case GVT is between the VRTs of events $e0$ and $e1$. The IAT to event $e1$ is the difference between the VRTs of the events $e0$ and $e1$, i.e., $(vrt(e1) - vrt(e0))$, but due to the simulators event ordering rules, no more events can arrive before GVT. Therefore, the probability that no event will occur before $e1$ is given by equation (7.5) where a and b are as shown in figure [7-2].

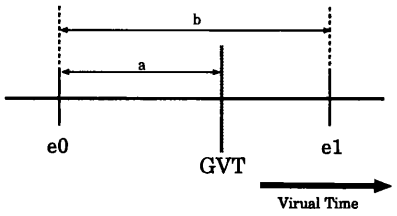

Figure 7-2: Diagram showing GVT between two event arrivals.

P(no events between e0 and e1)

$$
\begin{aligned}
&= P(\,(X \geq vrt(e1) - vrt(e0)) \;\mid\; (X \geq vrt(e0) - GVT)\,) \qquad (7.5)\\
&= \frac{P(\,(X \geq vrt(e1) - vrt(e0)) \cap (X \geq vrt(e1) - GVT)\,)}{P(\,X \geq GVT - vrt(e0)\,)}\\
&= \frac{P(\,X \geq (vrt(e1) - vrt(e0))\,)}{P(\,X \geq (GVT - vrt(e0))\,)}\\
&= \frac{P(\,X \geq b\,)}{P(\,X \geq a\,)}
\end{aligned}
$$

Since the value of GVT known at an executive could have been calculated some time previously, its value is likely to be less than the current SBT. For the probabilities to reflect the true situation, the value of SBT should be used in place of GVT, but this can not be done since the value of the SBT is not known. This problem is resolved by using a local estimate of the SBT in place of GVT in the calculation of the probability. This, along with the problems it introduces is explained in section 7.6.

7.3.3 Forecasting

Simply determining the probability that a particular IAT observed is likely to be correct is not enough to enable the event execution decision process to succeed. An event is only known to be correct once the GVT value known by the executive reaches the event's VRT. Since the GVT value known by the executive may be much lower than the SBT, it may be necessary

for an LP to execute many events without knowing if any of them are correct. This requires forecasting the future of the event stream beyond the first event encountered after GVT.

The probability of a series of events occurring can be found using conditional probabilities (7.6).

$$P(A \cap B) = P(A|B).P(B) \tag{7.6}$$

This says that if there were two events in the same stream, $e1$ and $e2$, and GVT was less than both of their VRTs, then the probability of $e1$ and $e2$ being in the correct order would be given by the probability that $e1$ was in the correct order multiplied by the probability that event $e2$ was in the correct order given that it is known that $e1$ is in the correct order. This can be extended to calculate the probability that a series of events are correctly ordered.

To calculate the probability of a series of events correctly, for each consecutive pair of IAT values in the series, iat_i and iat_{i+1}, the value of $P(iat_{i+1}|iat_i)$ would need to be known. In order to calculate this the correlation between the IAT values would need to be known.

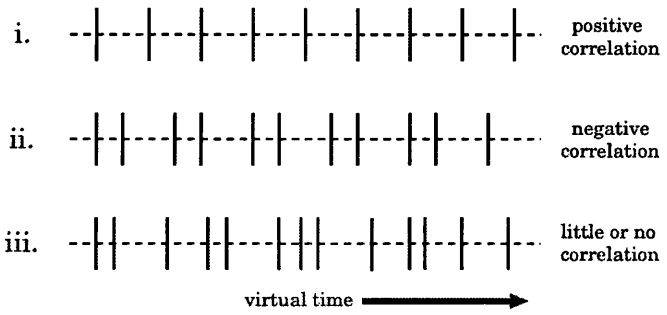

Figure 7-3: Event streams with different types of IAT correlation.

The stream of IATs will either be **(i)** positively correlated, **(ii)** negatively correlated or **(iii)** have little or no correlation; these three cases are depicted in figure [7-3]. Positive correlation means that given IAT_a, the chance that the value of the next IAT will be the close to IAT_a's value is high. Negative correlation means that given IAT_a the chance that the next IAT will not be close is high. In the no correlation case, the value of any IAT tells us little about the expected value of the next IAT.

The correlation between events could be determined using an auto-correlation function, but this would be expensive to calculate dynamically and could only be done after a large sample of IATs had been collected. I do not believe that the use of such a function will be of any great benefit. Notice that if the IATs are independent, then $P(A|B) = P(A)$, so equation (7.6)

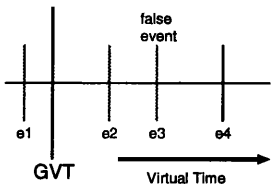

Figure 7-4: The false event problem. Even though event e3 is false, the system does not know that until it is annihilated and therefore will include it in the belief calculation as if it were correct.

reduces to (7.7) and this is what is used in BBS.

$$P(A \cap B) = P(A).P(B) \tag{7.7}$$

It is clear that not attempting to understand the correlation between IATs is another approximation. If it happens that the event stream has no correlation, then the calculation will be good. If the stream is positively correlated it is hoped that the other adaptive variables will adjust to keep the LP in question executing events slightly over-optimistically. If the stream is negatively correlated, some more errors are likely to be introduced in the calculation, but in this case it is likely that errors will only occur in small batches since the negative correlation means that if the IATs are small at one point, thus inducing early execution, soon after the IATs will be large thus leading to the slowing of event execution at this LP.

7.3.4 Probabilities and Beliefs

So far this section has looked at the probability of events being correct using forecasting based on event inter-arrival statistics. While these are the basic ideas that are used in belief based scheduling, I avoid the use of the word "probability" in the description of the algorithm as the number of approximations that are required to make this scheme work mean that the probability calculations are only really used to guide heuristics, rather than being strictly adhered to.

There are two major problems with using statistics and probability in this way, the first of which I call the *false event problem.* Consider the sequence of events shown in figure [7-4], where event $e3$ happens to be a false event. The belief based calculation will not know that $e3$ is in fact false until an anti-message arrives to annihilate it. Therefore, until that happens the contribution of $e3$ to the belief calculation at this LP will be the same as if $e3$ were correct. If it happens that $e3$ has the same VRT as an event that will replace it, then however good the inter-arrival based forecasting, the information gained from $e3$'s arrival will be incorrect.

The second problem is that the distribution of event inter-arrival times observed at an LP

may continually change throughout the simulation. Relying on forecasting based on out of date statistics will lead to poor decisions being made, so some method is required to determine when decisions based on runtime statistics are poor. Determining this is not easy particularly in the case where the decisions made are too conservative. When the system is being too optimistic, this is can be detected by an increase in rollbacks. Just because no rollbacks are occurring this does not mean that the system is being too conservative, indeed it may indicate that the system is perfectly optimised. Since it is easier to observe over-optimistic behaviour, the policy employed is to always be gradually increasing the optimism until over-optimistic behaviour becomes apparent; at that point the parameters are readjusted to make the system more conservative. Another problem with being too conservative at some LPs is that it can lead to rollbacks at other LPs when the overly conservative LPs eventually execute their events. Aiming to keep all LPs operating in a slightly over-optimistic way should help to prevent this type of throttling side effect.

The problem of detecting under-optimistic behaviour is not the only reason for attempting to keep the system working slightly over-optimistically. Since DTW uses lazy cancellation (see section 2.8) performance is not necessarily decreased by executing some events early.

The belief level for a series of events being in the correct order is found using (7.8) where the x_i's represent the IATs between the event VRTs, and t_0 represents the time difference between GVT and the timestamp of the first event at or after GVT, i.e., t_0 is **a** in figure [7-2] and x_0 is **b** in figure [7-2], and e_i is the candidate event.

$$\begin{aligned} belief &= \prod_{i=1}^{n} P(\text{no more events before event } e_i) \\ &= \frac{P(\,X \geq x_1\,)}{P(\,X \geq t_0\,)} \times \prod_{i=2}^{n} P(\,X \geq x_i\,) \\ &= \prod_{i=1}^{n} \frac{P(\,X \geq x_i\,)}{P(\,X \geq t_0\,)} \end{aligned} \tag{7.8}$$

7.3.5 Belief Level Adjustment

Deciding on the value of the execution belief level (EBL) is not simple. The most obvious value for EBL is 0.5 (i.e., execute an event if we believe there is a 50% chance of it being correct).

A problem is that in the scheme being described, the same EBL is used whether the event is the first event in the current stream since GVT, or if there has been a series of events with the candidate's belief level being calculated using (7.8). Since all IATs are assumed not to be correlated (see section 7.3.3), the actual belief level calculated may be quite low. Consider the

simple example where there are two events in the stream since GVT and that the current belief level in the first event is 0.7 and the calculated probability of the IAT between the events is also 0.7. Using (7.8) the belief in the second event is $0.7 \times 0.7 = 0.49$, which is less than 0.5. Therefore, even though the probabilities of both the IATs is 0.7 which appears good, the second event will not be executed at this time. This is one argument for having an EBL less that 0.5. Other reasons for having a low EBL are to keep the system running optimistically to take advantage of lazy cancellation and to allow extra parallelism to be exploited.

Instead of fixing a particular value, in BBS the EBL of each LP is updated dynamically during the simulation. At the start of the simulation the EBL values are set to a default initial value. During the simulation the value of the EBL at each LP is updated according to the rollback behaviour observed at the LP. The heuristic used in DTW to achieve this is described in section 8.2.

7.4 Multi-Event Execution

As noted in the introduction to this chapter, BBS schedules LPs to execute events rather than directly scheduling events to be executed. There is a distinction between the decision of whether an event should be executed, which is taken once the destination LP is scheduled, and when the LP itself should be scheduled. This distinction opens up the possibility to execute more than one event once an LP is scheduled. This ability will be referred to as *multi-event execution* (MEE).

MEE has been observed to be beneficial to the performance of conservative simulators; this was first pointed out to me by Chris Booth in 1996 and is emphasised by the performance of the *Critical Channel Traversing* algorithm Xiao *et al.* [75]. The main reasons for this are that cache locality is improved and that the scheduling costs are amortised over the execution of a number of events.

There is a major difference between a channel based conservative system (see section 1.1.2) and a Time Warp system. In the conservative system it is possible to determine absolutely which events are safe to execute in an LP's execution session. In a system without channels it is not generally possible to determine if an event is safe to execute. The belief calculation used by BBS does give an indication of the likelihood of an event being correctly ordered, so an MEE scheme can be implemented using this belief calculation.

It would be possible to have the belief calculation perform the task of determining when a scheduled LP should end its execution session. With this scheme an LP would be scheduled and continue to execute events until a candidate event is encountered that does not attain the EBL. Experiments with the implementation of BBS described in chapter 8 have shown that there are two problems with this policy. First, the belief calculations performed are not usually

very accurate. In the case where high belief levels are being calculated in error, allowing an LP to execute a large number of events based on this information is likely to lead to events being executed early. The second problem is that even if the belief calculation is working reliably, an execution session in this scheme will always end with either the LP running out of events to execute or in a failed belief calculation. It has been observed that if LP execution sessions usually end with an incorrect belief calculation, the cost of these wasted final belief calculations appears to impact on the performance of the system.

Instead the belief calculations are performed in conjunction with a counter that aborts the execution session after some number of events have been executed. The number of events executed before the execution session is aborted could be a simulation constant. The problem with using a constant is that the ideal value will be simulation specific and often LP specific. Due to this it is better to adapt the maximum event execution count to particular LPs as the simulation runs. The scheme used to achieve this in DTW is described in section 8.5.1.

7.5 LP Scheduling

The previous sections have explained how it is possible to determine if a candidate event has a good chance of being the next event that should executed by an LP. In this section the issue of which LPs should be scheduled for event execution is considered.

Since the decision of whether or not to execute a particular event is not taken until an LP is scheduled, the order in which LPs are scheduled will not necessarily affect the order that events are executed, since it is possible that once scheduled the decision is not to execute any events. If however it is often the case that the LP scheduled does not execute any events, it is likely that the simulator will be very inefficient. Also, if for any reason the belief calculations are allowing events to be executed early at some LP, the simulator could be more efficient if the LP scheduler prevented an LP that was experiencing rollbacks from receiving more than its fair share of CPU cycles.

Some possible polices for scheduling are considered are considered in section 7.5.1 and then the actual scheme used in BBS is described in section 7.5.2.

7.5.1 LP Scheduling Policies

The simplest LP scheduling policy that could be employed would be *round robin* scheduling, where each LP is repeatedly executed in turn. In most cases such a scheme would perform poorly due to often scheduling LPs that either have no events, or whose next event's belief level is below the EBL. Due to this round robin would not be successful in a general purpose simulator.

Another simple policy would be *lowest timestamp first* (LTF) scheduling that was described in section 2.2.2. The LTF scheduler would always schedule the LP with the lowest timestamp event even if this event could not currently be executed. This could lead to failing to take advantage of all the available parallelism. Suppose the LP selected by the LTF scheduler fails to execute even a single event. If no events arrive with timestamps less than that of the next event to be executed by this LP, the same LP will be scheduled again immediately. If the local estimate of SBT is not increased between the two execution sessions, the LP will again fail to execute any events. This process will continue until either a lower timestamped event arrives, an equal timestamped event arrives for another LP or the local SBT estimate increases enough for an event to be executed by the LP with the LTF event.

Note that it is possible that at any particular time there is no event on an executive that is safe to execute, but with LTF scheduling there is no chance of finding a safe event if the lowest timestamped event is not safe. Note also that the use of a real time delay before the execution of a low belief event, as is done by the previously mentioned algorithms, Srinivasan and Reynolds [60], and, Ferscha [17], is similar to this LTF based scheme. The main difference being that in the LTF based scheme described here, real time is taken up scheduling the same LP multiple times rather than simply suspending the executive.

The policy proposed by Som and Sargent [56] is more complex. In this the LP scheduled is the one whose next event has the highest probability of being correct. The algorithm proposed also only executes one event during each LP execution session, though could easily be modified to allow multi-event execution. The real difficulty with this scheme is that as the SBT increases, the probabilities of events held by each LP also changes. If no SBT estimation is performed the order in which LPs are scheduled may well not reflect the true likelihood of the events being correctly ordered; this is due largely to the approximations that have to be made in the belief calculations due to not knowing the correlation properties of the observed sequence of IATs. If on the other hand the SBT is estimated all the LPs would need to have their event probabilities updated each time the SBT estimate increased. Results suggesting that this algorithm was successful at reducing rollbacks (note that no claim was made of decreasing runtime) were presented in [56]. It should be noted that these results refer to the execution of a simulation model of the proposed scheduling algorithm, not to results from a simulator using the algorithm. Experiments performed for this thesis work suggest that such a scheme would not give good performance in the majority of simulations due to the high computation costs.

7.5.2 KSched Scheduling

The need for a scheduling algorithm that was computationally cheap and resulted in scheduling LPs with at least one event ready to execute led to a scheme based on one of the early execution throttling techniques. *Penalty based scheduling*, Reiher *et al.* [46] works by penalising an LP

that sends anti-messages. The penalty based scheme was described in section 6.2.1.

The scheduling algorithm used in BBS, which will be referred to as KSched, has two differences from the penalty based algorithm. A different policy is employed for determining when to suspend an LP, and the number of scheduler steps for which a suspended LP remains suspended is updated dynamically as the simulation progresses.

In the penalty based scheme an LP is suspended as soon as a rollback occurs at the LP. The logic behind doing this is that if an LP has just experienced a rollback, it has just received a late event and the chance of it receiving another late event in the near future is increased. Also, the fact that it experienced a rollback shows that it had earlier been given CPU cycles that might have been better used by some other LP.

There is an argument against this policy however. The opposing view is that if an LP has just rolled back, it is more likely than other LPs on the same executive to now be executing on the critical path, and therefore should now be given CPU cycles. Simply allowing the LP to execute forward freely could cause another problem however. If the rolled back LP is treated like other LPs on the same executive it could be given more CPU cycles than other LPs on the same executive, thus increasing the chance of missing the arrival of more events and thus suffering further rollbacks. The reason it might be given more cycles than other LPs is that it is now likely to have a number of events timestamped before the next event to be executed by any of the other local LPs.

This thinking led to the requirement that a rolled back LP should execute some small number of events after the rollback and then be penalised. The scheme employed simply removes an LP from the scheduler when the first candidate event in the LPs execution session fails to attain the required belief level. In the time between when an LP is suspended and when it is returned to the scheduler, it is likely that the SBT estimate will have increased, possibly by enough to allow one or more events to be executed in the LPs next execution session.

The second difference between KSched and the penalty based scheme is that, K, the number of scheduler steps that an LP remains suspended for is adjusted during the simulation run. Increasing K causes LPs to be suspended for a longer period. This increases the chance that one or more events will be ready to be executed during the LPs next execution session. However this also increases the chance that an LP will be woken late and thus trigger one or more rollbacks with events generated during its next execution session. Decreasing K has the opposite effect.

A modification to this scheme would be to suspend an LP for a number of scheduling steps proportional to the difference between the LPs execution belief level and the calculated belief for the event. While this idea has some parallels with the variable suspend times used in Srinivasan and Reynolds [60], and in Ferscha [17], I do not consider it to be particularly useful in this situation. The main reason for this is that the belief calculations are not in general

accurate enough to be used for making detailed decisions.

7.6 Estimating the SBT

As stated in section 7.3.2 it is useful to be able to estimate the value of the system base time in order to calculate event belief levels. Estimating the SBT locally to an executive is at best difficult. In a distributed system it is difficult to determine the behaviour of the function defining SBT update. Even if the values of GVT received by each executive form a linear function with respect to wall clock time, this does not mean that the SBT increases linearly between GVT calculations. Worse than this, it is possible that some property of the GVT algorithm could cause GVT values to be calculated that hide the true behaviour of the SBT increase function. Despite these difficulties experimentation has shown that it is beneficial to use an SBT estimate in conjunction with the event belief calculation.

The SBT can be estimated by first finding the rate of increase of GVT, then using this value to calculate the estimated SBT (eSBT). The value of the rate of change of GVT could be found in terms of wall clock time (7.9) or scheduler cycles (7.10); where a scheduler cycle involves scheduling an LP and executing a single event.

$$\frac{d}{dt}GVT \quad (7.9) \qquad\qquad \frac{d}{dc}GVT \quad (7.10)$$

The value of (7.9) could be estimated by dividing the increase in GVT in the last GVT round by the wall clock time taken for the round. The value of (7.10) could be estimated by dividing the increase of GVT in the last GVT round by the number of scheduler cycles that occurred in that GVT round. In both cases a better estimate could be found by using a linear approximation based on information collected over the last few GVT rounds.

If wall clock time is used, the value of eSBT could then be found using (7.11), where t represents the the amount of wall clock time that has passed since the last value of eSBT (eSBT_{old}) was calculated.

$$\text{eSBT} = \text{eSBT}_{old} + \frac{d}{dt}\text{GVT} \times t \quad (7.11)$$

If scheduler cycles are being used the value of eSBT could be calculated using (7.12), where c is the number of scheduler cycles since eSBT_{old} was calculated.

$$\text{eSBT} = \text{eSBT}_{old} + \frac{d}{dc}\text{GVT} \times c \quad (7.12)$$

It is not entirely clear which of these schemes should be used. Using (7.12) assumes that all event computations will take roughly the same amount of wall clock time. In most simulations

this will be the case, but in some models some events could take far more time to execute that others.

On the other hand, calculating (7.11) requires determining the current wall clock time (or at least how much wall clock time has passed since the last calculation). This can be difficult to do on many computer systems without making a system call to read the system clock; an operation that generally takes many CPU cycles.

Note that both schemes implicitly assume that SBT increases linearly, at least over short periods of time. While this may not be the case it is difficult to find any other approximation for the SBT increase function and experimentation with such a system in DTW has shown that this approximation is good enough in general. It is almost certainly possible to construct cases where the assumption of linear increase could cause problems, but since BBS uses a number of heuristics to determine what action to take the number of situations where this approximation is likely to cause problems to the scheduling scheme should be small.

In a system that employs a method to prevent thrashing behaviour the assumption that GVT increases linearly may not be too unreasonable, Fujimoto [45]. The assumption that SBT increases linearly is a little harder to justify, so any algorithm using the SBT estimate must be able to recover if the estimate provided is incorrect.

Other methods could be used for calculating the eSBT. In Som and Sargent [56], the scheme employed uses the minimum local clock of any LP on the executive, where the local clock is defined as the timestamp of the last message executed. Experiments performed for this thesis work suggest this to be a poor estimator, as the minimum local clock value is often close to or lower than GVT with all but the simplest of synthetic workload simulations.

One other scheme tried in DTW's BBS implementation increased the value of eSBT to the timestamp time of the lowest timestamped event on the executive each time the number of LPs suspended crossed some threshold value. This scheme worked well in some cases, though it is not currently the default algorithm used in DTW's BBS implementation (see section 8.8).

7.6.1 Incorrect Calculation

It has to be remembered that the eSBT value is only an estimate and will rarely equal the true SBT. The fact that the eSBT value is likely to be wrong has to be taken into consideration.

When used with BBS, a high eSBT value will cause event beliefs to be higher than they should be, causing more events to be executed early. A low eSBT value will cause event beliefs to be lower than they should be, thus throttling event execution. A low eSBT value will eventually be detected when a GVT value arrives that is greater than the current local eSBT value. A high eSBT value will be detected whenever an event arrives at the executive with a timestamp less than the eSBT value. Note that the arrival of a GVT value that is less than the eSBT value does not indicate that the eSBT value is too high. GVT is only a lower bound on

the current SBT.

7.6.2 SBT Estimate Limiting

A partial solution to the over estimation of the SBT is to prevent eSBT from increasing freely for the entire time between the arrival of GVT values. This could help to reduce the chance of runaway processes causing the sorts of problems described in section 4.8. Slowing the eSBT valve's advance will cause the event belief calculations to return lower values, thus slowly throttling the simulation. The slowing of eSBT advancement could be achieved in a number of ways:

- Cap eSBT to prevent it increasing passed a set value in the current GVT round.
- Scale each new eSBT value in an attempt to have eSBT trailing further and further behind the true SBT as the GVT round advances.

The scaling of eSBT values could either be done with a constant scaling factor, or with a function that decreases the rate of increase of eSBT the more wall clock time passes since the last GVT update. One problem with employing one of these schemes is that it adds another factor to interact with the belief calculations, possibly making it more difficult to determine why a particular rollback happened or why the system appears to be behaving more conservatively than expected.

7.7 Summary

This chapter has outlined the ideas behind the belief based scheduling (BBS) algorithm. The following chapter explains the implementation of the algorithm in DTW.

The aims laid out for the algorithm have been addressed as follows. The requirement for control of modes of operation that are detrimental to the performance of the system is provided by the belief calculations. These are calculations based on simple statistical forecasting to detemine if an event should be executed at the current time. The improved cache locality required is provided by multi-event execution (MEE), where more than one event can be executed in each LP execution session. MEE also has the benefit of amortising the per-event scheduling cost over all events executed in the LP execution session.

The belief calculations have added extra costs to the event scheduling system, but these costs are lower than in systems that attempt to use sophisticated statistical techniques, such as time series analysis. Also, the KSched LP scheduler uses a penalty based technique to avoid scheduling LPs that are unlikely to have events ready for execution.

Chapter 8

BBS Implementation

This chapter explains the Belief Based Scheduling (BBS) algorithm as it is implemented in DTW. The basic ideas used in the BBS algorithm were outlined in chapter 7.

This chapter is laid out as follows. Section 8.1 describes the representation used for the cumulative distribution function (CDF) held by each LP. Then sections 8.2 and 8.3 explain how the event belief levels are calculated and how they are updated when the local SBT estimate increases or when a rollback occurs. Section 8.5 explains how the multi-event execution code is implemented and section 8.6 explains how the CDF of each LP is updated to reflect changes in the distribution of IATs observed in the event stream. The final two sections, 8.7 and 8.8 explain the implementation of the LP scheduler and the SBT estimation algorithm.

8.1 CDF Representation

This section explains how the representation of the cumulative distribution function (CDF) used in BBS is implemented in DTW. A compact representation of the CDF is required since each LP maintains its own CDF and there could be a large number of LPs per executive. While it would be useful if a simple mathematical function could be used to generate the CDF values when required, constructing such a function is not possible in most cases. Instead a histogram is maintained from which approximate CDF values can be obtained. In most cases looking up the value from the histogram will be faster than calculating the CDF value from a mathematical function.

The CDF histogram is represented by a vector of floating point numbers. This vector has eight elements; a size that was found to be effective experimentally. Each element, which will be described as a bucket, represents a range of IAT values. The value in each bucket represents the value of the CDF, $F(x)$, where x is the greatest valued IAT represented by the bucket. Figure [8-1] shows how a continuous CDF, $F(X)$, is represented in vector form. Note that

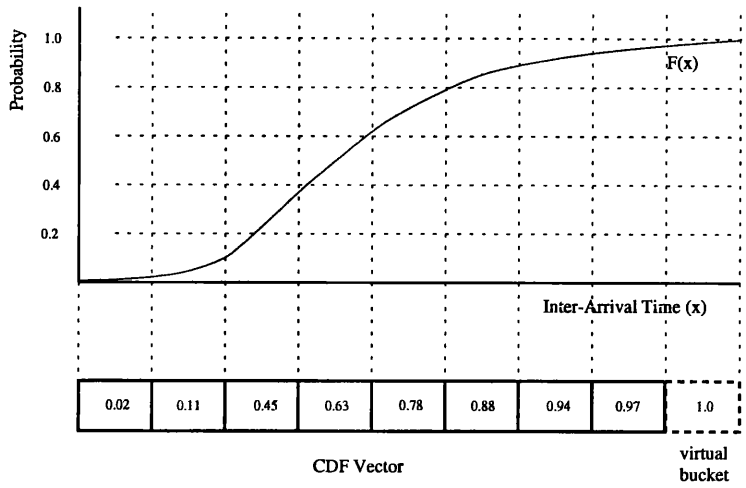

Figure 8-1: Diagram showing how a CDF F(X) (continuous in this case) maps to the vector representation.

the mapping from the CDF is actually to one more bucket than the number of elements in the vector. This is because the value in the final bucket, the "virtual bucket" in the diagram will always be 1.0 and therefore does not need to be stored.

8.2 Calculating Beliefs

The probability of a particular IAT occurring is found from the CDF vector using linear approximation. First the argument to the CDF is calculated using (8.1), where iat is the IAT under consideration.

$$x = \text{iat} - \epsilon \qquad (8.1)$$

$$idx = \left\lfloor \frac{x}{bucket\_size} \right\rfloor \qquad (8.2)$$

The reason for using the correction value ϵ was explained in section 7.3.1. Then the index of the bucket representing the value x is found using (8.2). The IAT probability is then calculated using (8.3) where, t_0 is the "distance" of x into the indexed bucket, i.e., $t_0 = x - idx \times bucket\_size$, $b1$ is the value held in bucket idx and $b0$ is the value held in the previous bucket, or is 0.0 if that value of idx is zero. This situation is depicted in figure [8-2].

$$P(\, X < iat \,) \approx P(\, X \leq iat - \epsilon \,) = b0 + (\, b1 - b0 \,) \times \frac{t_0}{bucket\_size} \qquad (8.3)$$

Using (8.3), the probability that no events should occur before the candidate event is cal-

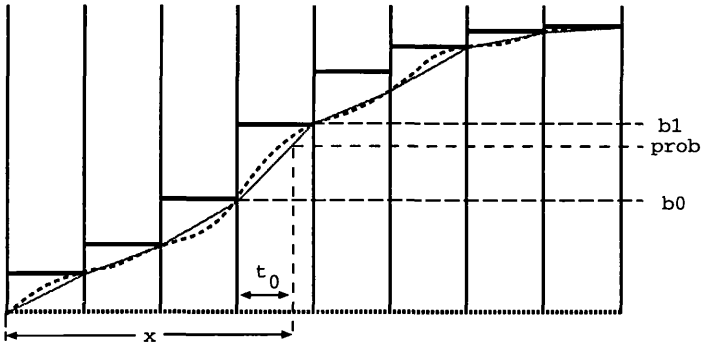

Figure 8-2: Interpolating probability approximations from the CDF vector.

culated from (8.4), where t_0 is defined as in (8.3).

$$\begin{aligned} P(\, X \geq iat\,) &= 1 - P(\, X < iat\,) \\ &\approx 1 - P(\, X \leq iat - \epsilon\,) \\ &\approx 1 - b0 - (\, b1 - b0\,) \times \frac{t_0}{bucket\_size} \end{aligned} \tag{8.4}$$

As described in section 7.3.4, BBS assumes that IATs are not correlated and therefore uses (8.5) to calculate the belief level in a series of events being correctly ordered, where the x_i's represent the ϵ corrected IATs between the events in the stream, and δ_0 is the difference between eSBT and the timestamp of the first event in the stream at or before eSBT.

$$belief = \prod_{i=1}^{n} \frac{P(X \geq x_i)}{P(X \geq \delta_0)} \tag{8.5}$$

This calculation is performed incrementally using two variables maintained for each LP. The values keep are, *base_prob*, the probability of the first event after GVT being correct and *running_total*, which represents a belief level in a series of events. If n events, $e_1, ..., e_n$, have been executed that have timestamps greater than eSBT, then the value of *running_total* is defined by (8.6).

$$running\_total = \begin{cases} 1.0 & \text{if } n < 2, \\ \sum_{i=2}^{n-1} P((vrt(e_i) - vrt(e_{i-1})) - \epsilon) & \text{otherwise.} \end{cases} \tag{8.6}$$

This accounts for all the executed events apart from the first. The probability of the first event being correct is calculated using (8.7) where x_1 is the IAT between the last event timestamped

at or before eSBT and the next event in the stream, and δ_0 is the virtual time interval between the last event timestamped at or before eSBT, and eSBT. Note at the start of the simulation the time of the last event to be executed is set to 0.0.

$$base\_prob = \frac{P(X \geq x_1)}{P(X \geq \delta_0)} \tag{8.7}$$

The belief level in an event being correct is then calculated using (8.8) where *iat_prob* the probability of no more events arriving between the candidate event and the last event in the stream.

$$belief = iat\_prob \times running\_total \times base\_prob \tag{8.8}$$

The value of *iat_prob* is stored in event e_i's data structure when event e_i is executed. This allows the value of *running_total* to be updated incrementally as explained in section 8.3.

8.3 Updating Beliefs

This section explains how the variables used in the belief calculations are updated as the simulation runs. First the way in which the variables are updated to reflect the passing of virtual time is explained, then how the variables are updated in the case of a rollback is explained.

8.3.1 Time Based Updates

A local estimate of the SBT is used to increase the belief in events as wall clock time passes. The reasons for doing this are explained in section 7.3.2 and the implementation of the SBT estimating code described in section 8.8.

As the value of eSBT reaches the timestamp of an executed event it is assumed, for the sake of the belief calculation, that this event is committed. Note that it is not known for sure that the event is committed until GVT reaches the event's VRT. Since the event is assumed to be committed its probability of being correct is now assumed to be 1.0. This has an effect on the belief level of later events that is calculated using (8.5).

The beliefs are updated by updating the values of *running_total* and *base_prob*. The value of *running_total* is updated by dividing out $iat\_prob(e_{old})$, where e_{old} is the event whose timestamp is now less than or equal to eSBT and $iat\_prob(e_{old})$ is the IAT probability that was stored in e_{old}'s record structure when it was executed. The value of *base_prob* is recalculated using (8.7), where δ_0 and x_1 are as defined in section 8.2.

Using eSBT to update the event beliefs carries the danger that the eSBT value may be too high and cause events to be executed that a correctly operating belief calculation would not

have allowed. To reduce this problem, when an LP is scheduled and finds that eSBT has decreased since it was last scheduled, the values of *base_prob* and *running_total* are recalculated using (8.5). The value of eSBT is reduced whenever it is proved to be to high, as explained in section 8.8.

Note that one problem with the incremental update scheme is that rounding errors are introduced to the value of *running_total* as new values are multiplied in and divided out. To reduce the effect of errors due the value of *running_total* erroneously increasing, its value is prevented from ever exceeding 1.0. In the case where the value of *running_total* is erroneously low, the LP will become more conservative, until there are less than two events at the LP that have been executed, but have timestamps greater than eSBT. At this point the value of *running_total* is reset to 1.0 thus removing any error that may have been introduced.

8.3.2 Rollback Based Updates

When an LP rolls back, the value of *running_total* has to be updated. If all the events with timestamps greater than the eSBT value when the LP was last scheduled are rolled back, then the value of *base_prob* is reset to 1.0.

The value of *running_total* is updated by dividing out the probability approximations for all events rolled back. Again, this is done using the values that were stored in the event records when the events were executed. The same problems with rounding error can occur in this operation as were described in section 8.3.1 and the same mechanisms are used to prevent this causing problems.

8.4 Execution Belief Level

The previous sections have shown how a belief level can be calculated for a particular event. The event is only executed if this belief level exceeds a threshold value described as the *execution belief level* (EBL). As described in section 7.3.5 an LP's EBL is adjusted according to the rollback behaviour of the LP. The greater the EBL the more conservatively the LP will behave. The lower the EBL the more optimistically the LP will behave.

The EBL value for each LP ranges between the constant values *min_BBS_EBL* and *max_BBS_EBL*. If LP_i experiences a rollback and (8.9) is true then EBL_i is increased using (8.10).

$$\text{EBL}_i < \text{max\_BBS\_EBL} \quad (8.9) \qquad \text{EBL}_i = \text{EBL}_{i_{old}} + \text{inc\_BBS\_EBL} \quad (8.10)$$

If during a fossil collection session it is found that LP_i has experienced no rollbacks since the last fossil collection session and (8.11) is true, then EBL_i is updated using (8.12).

$\text{EBL}_i > \text{min\_BBS\_EBL}$ (8.11) $\text{EBL}_i = \text{EBL}_{i_{old}} - \text{dec\_BBS\_EBL}$ (8.12)

dec_BBS_EBL	inc_BBS_EBL	min_BBS_EBL	max_BBS_EBL
0.02	0.01	0.02	0.99

Table 8.1: Values used for the event belief level constants.

The values of the EBL constants used in DTW are shown in table [8.4]. These were determined to be effective experimentally. Note that the value by which the EBL is decreased by is greater than the value by which it is increased. Probably the best way to explain this asymmetry is that there is a need to quickly gain control of an LP's event execution after a rollback occurs. As for the slower increase in the LP's EBL, this might be explained by the need not to release the added constraints on a recently rolled back LP too quickly. Either way the adjustment values are quite small, so any one adjustment has little effect. All LPs start with an EBL of 0.3.

8.5 Multi-Event Execution

This section describes the actions taken when an LP is scheduled. First the event processing loop is described in section 8.5.1, then how the maximum number of events to be executed in the event loop during a single LP execution session is explained in section 8.5.2.

8.5.1 Event Execution

The event execution loop is contained in the LP event execution function. This is called with an LP when the LP is scheduled for an execution session. When the event execution function is called for LP_i the actions described below are performed. Note that at most $nevents_i$ are executed in the session; this is explained further in section 8.5.2.

Within the event loop, *count*, the number of events executed in this execution session is maintained. At the start of the loop, b_i the belief in the event at the head of LP_i's event queue is calculated. Then, if $b_i \geq EBL_i$, the candidate event is executed. Some additional house keeping is performed to maintain the belief calculation; this was explained in section 8.2. Next if $count < nevents_i$, the thread of execution returns to the start of the event execution loop and if there is another event in the event queue, this is made the next candidate event.

Otherwise, if $b_i < EBL_i$, or if $b_i \geq EBL_i$ and *count* equals $nevents_i$, the event loop is exited. The event loop is also exited if an attempt to select a candidate event fails due to the

queue becoming empty.

The execution function returns the boolean value *false* if the first event to be executed fails to reach the execution belief level. The boolean value *true* is returned otherwise. The return value is used by the LP scheduling function; this will be described in section 8.7.1. Also, a flag is set if the event loop is exited due to *count* reaching $nevents_i$. This flag is considered when the decision is made as to whether the value of $nevents_i$ should be adjusted and this is explained in section 8.5.2.

8.5.2 Adjusting MEE

The value of *nevents* used in the event execution loop is updated periodically. This is done in the LP's fossil collection function if one or more events have been fossil collected in the current function call.

The value of *nevents* is decreased if its value is currently greater than 1 and at least one of the following is true:

- At least one event has been rolled back since the last fossil collection.
- The MEE bound has not been reached since the last fossil collection.

The value of *nevents* is increased if neither of the above are true and the current value of *nevents* is less than the constant value *MEE_max*. *MEE_max* is set to 5 in the current implementation.

8.6 CDF Maintenance

This section explains how the CDF vector is built and maintained. It explains how the initial CDF is constructed and how incremental updates are performed to both refine the representation and to adapt the representation to changes in the distribution of IATs appearing in the event arrival stream.

8.6.1 Building the CDF

Each LP has its own CDF vector representing the distribution of IATs in the event stream seen by this LP. The initial CDF is constructed by first building a PDF and then building the CDF from this PDF. The PDF is represented as a vector of floating point numbers with the same number of elements as the CDF being constructed. The PDF is constructed from a sample of IATs.

The sample of IATs used to construct the initial PDF is collected at the end of the simulation initialisation phase (see section 2.7). This is done by examining the event queues of all the LPs

on the executive. At this point the queues hold all the start events (see section 4.1). Since the PDF to be constructed is supposed to be particular to an LP it would be best to just sample events from the queue of the LP that a particular PDF is being constructed for. Experimentation has shown that this technique is poor since the number of start events held by each LP is small, and therefore the sample size is small. Instead a single PDF and CDF is constructed using the IATs of all the start events on the executive and its values are copied to the CDF vectors of each of the local LPs. This is not ideal but it is achieved without user intervention; one of the scheduler requirements listed in section 7.1. The individual CDFs will adapt themselves to the LP's event stream IAT distribution as the simulation runs (see section 8.6.2).

The first stage in building the PDF is to find *max_initial_value*, the maximum value of the IATs in the sample. Then *bucket_size*, the interval of time spanned by an element of the PDF and CDF vectors is found using (8.13).

$$bucket\_size = \left\lfloor \frac{max\_initial\_value}{(num\_buckets + 1) - \sigma} \right\rfloor \tag{8.13}$$

The addition of one to the number of buckets is done to take account of the virtual bucket. The value σ is used in (8.13) to make the vector larger than is required by a distribution whose maximum value is *max_initial_value*. This is done because the vector is updated as the simulation runs and making the vector a little larger has been observed to help prevent some vector resizing when larger IATs are observed early in the simulation. Currently σ is set to 0.5. The result of (8.13) are depicted in figure [8-3].

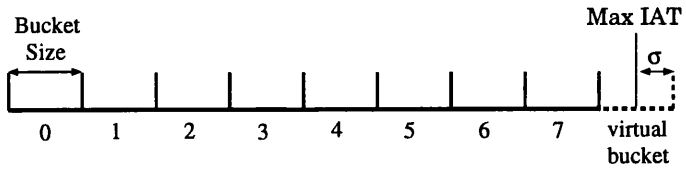

Figure 8-3: Setting the CDF vector bucket size.

Now the PDF is constructed from the set of initial IAT values S using (8.15), where $I(i)$ is the set of values defined by (8.14) and *init_ss* is the initial sample size, i.e., the number of IAT values in the sample.

$$I(i) = \left\{ x \in S : \left\lfloor \frac{x}{bucket\_size} \right\rfloor = i \right\} \tag{8.14}$$

$$pdf_i = \frac{|I(i)|}{init\_ss}, \quad \forall\, i \in \{0, ..., num\_buckets - 1\} \tag{8.15}$$

The CDF is now constructed from the PDF using (8.16).

$$cdf_i = \begin{cases} \sum_{j<i} pdf_j & \text{if i} < num\_buckets, \\ 1.0 & \text{otherwise.} \end{cases} \tag{8.16}$$

8.6.2 Dynamic CDF Update

One of the scheduler requirements listed in section 7.1 is that the scheduler should adapt to changing simulation behaviour as the simulator runs. This is done in part by adapting the CDF to changes observed in the distribution of IAT values.

This could be done by collecting a sample of IATs and using these to construct a new PDF and CDF as explained in section 8.6.1. The problem with doing this is that a large sample of IATs could be required to build a PDF that accurately models the distribution. Instead an incremental update scheme is employed that is executed whenever events are fossil collected. The incremental calculation produces a CDF that approximates the true CDF; considering the number of other approximations that have to be made the fact that the CDF itself is only approximated is not felt to be a problem.

Incremental CDF Update

The incremental update scheme uses a dummy sample size value, ss. This is used to represent the number of IATs that would have been used to build the CDF if it were being built from scratch. This value is important as it controls how quickly the values held in the CDF vector adapt to the changing event arrival pattern at the LP. If the sample size is large, the CDF will adapt slowly to changes. If the sample size is small, the CDF may not contain information on seasonal changes to the arrival pattern. In the current implementation ss has the constant value 20. It may be preferable if this value adapted to pick up seasonal changes in the event stream, but ways to achieve this are yet to be investigated.

An event is only fossil collected when it is known to be committed. At this point iat, the inter-arrival time from the timestamp of previous event in the stream to the timestamp of the current event being fossil collected could be added to the CDF representation using (8.17).

$$cdf_i = \frac{cdf_{i_{old}} \times ss + \gamma}{ss + 1}, \quad \text{where} \quad \gamma = \begin{cases} 0 & \text{if i} < \left\lfloor \frac{iat}{bucket\_size} \right\rfloor, \\ 1 & \text{otherwise.} \end{cases} \tag{8.17}$$

In fact in DTW this process is modified so that more than one new value is incorporated at

the same time. To do this, the integer vector v is constructed using (8.19), where $I(i)$ is the set defined by (8.18) and S is the set of IATs collected in the current fossil collection session.

$$I(i) = \left\{ x \in S : \left\lfloor \frac{x}{bucket\_size} \right\rfloor = i \right\} \tag{8.18}$$

$$v_i = \left| I(i) \right|, \quad \forall\, i \in \{0, ..., num\_buckets - 1\} \tag{8.19}$$

The CDF is then updated using (8.20) where ns is the count of the number of IATs in the current sample, i.e., $ns = |S|$.

$$cdf_i = \frac{1}{ss + ns} \times \left(cdf_{i_{old}} \times ss + \sum_{j=0}^{i} v_j \right) \tag{8.20}$$

CDF Resizing

The range of values represented by the CDF vector may require changing during the simulation. This could be necessary if one of the following is observed:

i) The latest IAT value is greater than the largest IAT represented in the CDF.

ii) The difference between the values held in the last two buckets of the CDF is tending to zero.

Case (**i**) shows that the CDF vector does not currently represent the whole of the distribution. Case (**ii**) shows either that one of the larger IAT values that has been observed has not been observed for some time, or that the range of values represented in the vector was too large for this LP to start with. In the case that the larger IAT has not been observed for some time, the CDF vector is resized since the distribution of IATs in the event stream may have changed and no more large valued IATs may be observed.

The CDF range is adjusted by first setting the value of *new_top_val*, using equation (8.21) in case (**i**) and using equation (8.22) in case (**ii**), where σ is defined as it was in equation (8.13).

$$new\_top\_val = new\_iat \tag{8.21}$$

$$new\_top\_val = (num\_buckets - \sigma) \times old\_top\_val \tag{8.22}$$

Here, *new_top_val* represents the largest IAT value represented in the CDF. The CDF is then reconstructed using (8.23) where cdf_i is the new value of the ith element of the CDF and $cdf_{i_{old}}$

is the old value of the ith element of the CDF.

$$cdf_i = \frac{cdf_{i_{old}}}{old\_size} \times \left\lfloor \frac{new\_top\_val}{(num\_buckets + 1) - \sigma} \right\rfloor \tag{8.23}$$

Note that in the current implementation of BBS, the first CDF bucket always starts from an IAT value of zero. This is not ideal for distributions of IATs where the smallest IAT observed in much greater than zero. It may be possible to improve the accuracy of the CDF's vector representation by defining an offset index value and having the CDF vector's first element represent IATs of some value greater than zero.

8.7 The LP Scheduler

This section describes the DTW implementation of the KSched scheduling scheme outlined in section 7.5.2.

8.7.1 Dual Queue Implementation

Two priority queues are used in the KSched implementation. The main *scheduling queue* is a heap data structure ordered by the timestamp of the next event waiting to be executed at each LP. As described in section 2.9, each LP has its own event queue making it easy to determine the timestamp value of the next event for each LP. An LP is removed from the scheduling queue before its execution session begins and is reinserted into the queue when its execution session ends. An LP's position in the queue could also change when an event arrives for the LP with a timestamp lower than the current lowest timestamped event held by the LP. This is achieved by a simple sorting procedure, with the LP being "bubbled" up the queue in log_2 steps. Note that if LPs were returned to the scheduling queue after executing a single event in each execution session, this would result in LTF scheduling.

The second queue is the *suspend queue*. An LP is placed into the suspend queue if the belief calculation for the first event in its execution session fails to reach the execution belief level required for this LP (see section 8.5.1). The suspend queue is a simple FIFO queue. LPs are placed into the queue along with $rc_{current}$, the current value of the scheduler round counter; a value that is incremented by one after each LP execution session.

After each LP execution session the round counter held with the LP at the top of the suspend queue, rc_{top}, is inspected. If the condition stated in (8.24) is true, where K is as defined in section 7.5.2, then the LP at the top of the suspend queue is returned to the main scheduler queue.

$$rc_{top} + K < rc_{current} \tag{8.24}$$

An LP may be woken early. This happens if the timestamp of a new event inserted into the LP's event queue is less than the timestamp of the previous lowest timestamped event in the LP's event queue. In this case the belief in the new event will likely be higher than that of the previous lowest timestamped event for this LP, thus prompting the LPs early return to the main scheduling queue.

8.7.2 Adjusting K

The value of K has several effects on the behaviour of the system. These are discussed in section 7.5.2. In the DTW implementation of BBS, K is adjusted when a new GVT value arrives at the executive.

A simple heuristic for adjusting K has proved to be effective. If the eSBT calculation (see section 8.8) has been found to be too high since the last GVT update, the value of K is increased by one. Alternatively, if some number of GVT updates occur without eSBT having been found to be too high and K is currently greater than one, then K is reduced by one. Currently, five GVT updates without a high eSBT value being detected must occur before the value of K is decreased.

The logic behind this heuristic is that increasing the value of K reduces the chance that the eSBT increase (see equation (8.28) in section 8.8) will be triggered in error. Other heuristics based on the number of rollbacks observed and on how often recently woken LPs can execute an event have been tried, but none has worked as reliably as the simple one presented.

8.7.3 GVT Implications

The use of the suspend queue required the local RT calculation mechanism (section 5.5) to be modified from that used with a LTF scheduler. This is because the value of the VRT of the event currently being executed no longer necessarily represents the lowest active timestamp on the executive. With BBS, the value of the LQT of all suspended LPs also has to be taken into account in the RT calculation. This is done by maintaining a priority queue containing the same LPs that are currently in the suspend queue, sorted by the LQT of each LP. The value passed to the RT calculation is calculated as the minimum of the next event time of the LP at the head of the main scheduling queue and the next event time of the LP at the head of the priority queue containing suspended LPs. This value is then used along with the transient message time to calculate the new RT value.

8.8 SBT Estimation

The SBT is estimated using the scheduler round based increase scheme described in section 7.6, where the rate of increase of GVT is calculated from (8.25) where gss is the sample size and gvt_i is an element of a vector holding the last gss GVT values to arrive at the executive. A gss value of 5 is used currently.

$$\frac{d}{dc}GVT = \frac{1}{gss}\sum_{i=0}^{gss} gvt_i \tag{8.25}$$

Instead of using equation (7.12) to calculate new values of eSBT, the eSBT value is calculated incrementally using equation (8.26). This calculation is performed before each LP is scheduled; note this does not mean it is calculated before each event is executed since more than one event may be executed in an LP execution session.

$$\text{eSBT} = \text{eSBT}_{old} + \frac{d}{dc}GVT \tag{8.26}$$

Other than its incremental update (8.26), the value of eSBT is adjusted if one of two conditions arises. The first of these is if an event arrives at the executive with a VRT less than the current eSBT value. In this case the value of eSBT is clearly greater than the true SBT, so the value of eSBT is reset to the value of the new event's VRT, i.e., eSBT is set using (8.27) where e_{new} is the newly arrived event.

$$\text{eSBT} = vrt(e_{new}) \tag{8.27}$$

The other case where eSBT is adjusted is when a local condition arises that makes it seem likely that the eSBT value known currently is too low. To determine if this is the case, a test is made when an LP is suspended to determine if the number of LPs suspended is greater than or equal to the current value of K (see section 7.5.2). If this is the case, the eSBT value is increased using (8.28) where e_{low} is the event with the lowest VRT of any active event on the executive.

$$\text{eSBT} = \frac{\text{eSBT}_{old} + vrt(e_{low})}{2} \tag{8.28}$$

In some cases this will result in eSBT being increased in error. The adjustment of **K** (section 8.7.2) should help to prevent the error case occurring too often. Experimental evidence suggests that eSBT errors caused by using (8.28) are less detrimental to performance than the effect of excessively low eSBT values that can occur without it.

The current implementation increases eSBT linearly after each LP execution session. None of the eSBT limiting schemes described in section 7.6.2 are currently used. While some sort of limiting scheme would make sense in a general purpose system, they will effect the behaviour of the system. I did not feel that attempting to understand the added behavioural complexity that comes with using one of these schemes would aid the understanding of BBS in general, so benchmarking experiments using such schemes remains to be done.

8.9 Summary

This chapter has explained the implementation of Belief Based Scheduling (BBS) in DTW. The representation of the CDF needed for performing the belief calculations was explained along with the methods used to maintain it. Most calculations are performed incrementally so that new information can be added and old information removed from the CDF as soon as possible. Also explained was the way in which the event belief levels are calculated from the CDF and how the execution belief level is adapted to reflect the recent rollback behaviour of each LP.

The DTW implementation of KSched LP scheduling was also presented. This provides a simple way to avoid performing belief calculations on events that are likely to fail the execution belief level test. Performance results for DTW using this BBS implementation are given in chapter 9.

Chapter 9

Testing

This chapter presents the simulation models implemented to test and evaluate DTW, along with some performance results obtained using these models. The main purpose of the performance analysis is to compare the performance of DTW using the BBS scheduler (see chapter 8) with DTW using an LTF scheduler.

Examples are shown for cases where LTF scheduling is known to perform well so that the impact of BBS in these cases can be observed. Also, cases where an unconstrained LTF scheduler performs poorly are presented along with the BBS results for the same simulations. Many of the performance graphs referenced in this chapter can be found in Appendix A.

Also presented in this chapter are statistics for the number of acknowledgement messages required to calculate GVT in the test simulations and a simple analysis of why the ratio of simulation messages and acknowledgement messages varies from simulation to simulation. Finally, the results are analysed and their implications discussed.

9.1 Simulation Models

Two simulation frameworks have been used in testing. The first, **SimLoad**, is a synthetic workload model with some similarities to the **PHold** synthetic workload model, Fujimoto [23]. This is a configurable workload model that enables the testing of many properties of real simulation models in a simple, easy to understand environment. It also provides hooks to allow more complex code to be added to the basic synthetic workload model. The second model is **Pucks**, a simulation model based on an implementation of the *distribution list algorithm*, Steinman and Wieland [67]; a distributed algorithm that distributes moving object equations of motion to other moving objects in the system. The **Pucks** simulator models circular pucks sliding on frictionless bounded surface where all collisions between pucks and between pucks and the boundary are modelled as being perfectly elastic.

9.1.1 SimLoad

SimLoad provides a framework for modelling the behaviours of simulations in a way that is easy to configure and is easy to understand. Hooks are provided for adding code in several locations. Without code additions, **SimLoad** acts much like the **PHold** model, Fujimoto [23], in that it can be configured using input data to set parameters that adjust some aspects of the behaviour of the workload. The adjustable parameters in the base model include:

- **initial event count**
  - the number of events generated by each LP during initialisation.
- **event execution load**
  - the number of state updates performed each time an event is executed.
- **event spread**
  - the number of LPs that any particular LP will send events to. This is configured so that if LP_i can send a message to LP_j, then LP_j can also send a message to LP_i.

In addition hooks are provided so that functions that perform the following actions can be attached to the system:

- **initial event generation function**
  - this allows more flexibility in the types of events generated initially.
- **message passing function**
  - this allows the message passing behaviour of each LP to be specified more accurately than if *event spread* is used.
- **event execution extension function**
  - this is called on event types other than the standard synthetic load events generated by the base system. This is used to perform special execution of event types originating from functions attached to the previous two hooks.

A model can either be constructed homogeneously, with each LP behaving in a similar way, or heterogeneously, with different LPs having different behaviours. A script is used to generate the input file with one entry per LP, which can either be modified by hand or by adjusting some parameters within the generating script in order to change the behaviours of particular LPs.

Much of the testing with **SimLoad** has been performed using the base system, but results are presented in this chapter for four specific workload configurations. The simulation models used are described using the names **BeRisky**, **Echo**, **Torus** and **Pipeline**. Apart from **BeRisky**, all these models use at least one function hook to specialise their behaviour. These models are described in the following sections.

9.1.2 BeRisky

In the **BeRisky** simulation model, first presented by Bellenot in [4], none of the LPs communicate with each other; they only generate self events. The effect of this is that no rollbacks can ever occur. While no real parallel simulation is likely to behave in this manner, **BeRisky** is an interesting test case for a PDES system. Experiments with **BeRisky** can show whether a simulation system manages to exploit the parallelism available in the model, thus highlighting possible problems that can occur when throttling algorithms use inappropriate throttling parameters.

9.1.3 Echo

The **Echo** simulation model is designed to induce the echo failure mode described in section 4.8. The DTW **Echo** model is different from the one presented by Lubachevsky and Weiss in [37]. The version presented here uses the difference in rates between self events and external events to induce echo. Another difference is that unlike the version presented in [37], this version is not restricted to a fixed number of LPs.

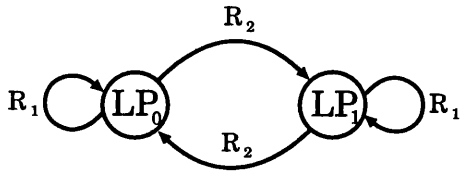

Figure 9-1: Diagram showing how events are exchanged in a two LP **Echo** configuration. Each LP generates self events with timestamp increments determined by rate R_1. At any point, a single external event exists in the system, with the arrival of an external event causing the generation of a new external event with a timestamp increment determined by rate R_2.

Each LP sends events to itself and generates these events at rate R_1. At the start of the simulation each LP generates a self event with VRT $\frac{1}{R_1}$. When a self event is received a new self event is sent with a VRT increment of $\frac{1}{R_1}$. Also at the start of the simulation, LP_0 sends an external event to LP_1 with VRT $\frac{1}{R_2}$. On receiving an external event an LP generates a new external event and sends it to the LP with the successive identifier using a modular wrap, i.e., in a two LP system (see figure [9-1]) LP_0 sends to LP_1 and LP_1 sends to LP_0 [1]. The timestamp of the new event is set using an increment function that increases the timestamps at rate R_2.

[1]Note that using the Echo workload with more than two LPs gives a system similar to the Rings workload described by Panesar and Fujimoto in [45]. I do not see the need to differentiate between the 2 LP and greater than 2 LP models since they both induce echo.

By adjusting R_2 relative to R_1 the chance of each successive external event arrival causing a rollback can be adjusted. The lower the value of $\frac{R_1}{R_2}$ the more events will be rolled back by each external event. The usual way of configuring an echo model is to place one LP on each executive; see Lubachevsky and Weiss [37], Panesar and Fujimoto [45] and, Srinivasan and Reynolds [60].

The tests presented in section 9.2.2 use a fixed value of 1.0 for R_1 and the value of R_2 is varied between 1.0 and 128.0. This results in self events being generated with timestamp increments of 1.0. External events have timestamp increments of between 1.0 and 0.0078 (1/128.0).

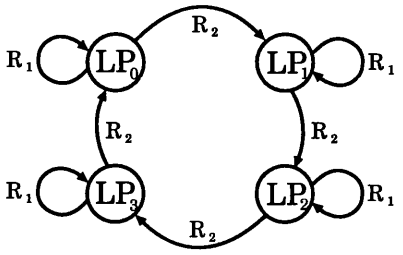

Figure 9-2: Diagram showing how more than 2 LPs can be used in a simulation using the **Echo** model. In this case 4 LPs each generate self events and pass an external event on to the next LP in the cycle each time an external event is received from the previous LP.

9.1.4 Pipeline

The **Pipeline** workload model is another model created to induce unstable behaviour in an unthrottled Time Warp system. Figure [9-3A] shows an example of a pipeline of LPs, with all but the first fed both by the previous LP in the pipeline and by another LP not in the pipeline. If events arriving from the previous LP in the pipeline have smaller timestamp increments than events arriving from the non-pipeline LPs, it is likely that rollbacks will be induced by the arrival of the in-pipeline events. Figure [9-3B] shows a simpler model where events from the non-pipeline sources are modelled by self events at each of the LPs in the pipeline. It is this model that is used for the **Pipeline** workload.

Initially LP_0 generates one self event and one external event, while all other LPs generate one self event. Each time a self event is executed by LP_0, it generates a new self event and a new external event. All other LPs generate an external event on the arrival of an external event and a self event on the arrival of a self event. Suppose that LP_{i+1} executes three events before the arrival of m_k, the first event from LP_i. If all the events executed by LP_{i+1} have VRTs less

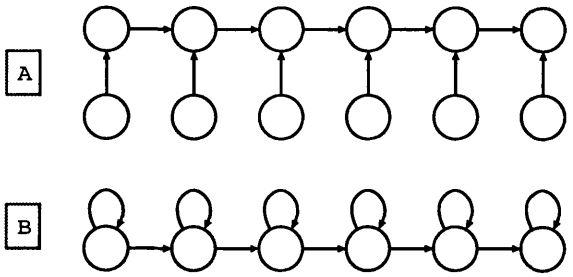

Figure 9-3: Diagrams showing the message passing model for **Pipeline**. Diagram A shows events arriving at LPs in the pipeline from LPs not in the pipeline. Diagram B shows a simplification of this system, where the events from non-pipeline LPs are modelled by self events generated by the LPs in the pipeline; this is the model used by the **Pipeline** workload.

than that of m_k, then no rollback will occur. If on the other hand, all the events executed have VRTs greater than $vrt(m_k)$, then all three events will be rolled back. Therefore the likelihood that any external event will cause a rollback can be adjusted by changing the lifetimes of the external events or by adjusting the rate at which self events are generated.

In the **Pipeline** experiments that results are presented for in section 9.2.3, the lifetime of all external events is set at 1.0 and the VRT of the self events calculated using equation (9.1), where **i** is the identity of the LP, an integer in the range [0, |LPs| -1], m_j is the jth self event to be executed by LP_i and *scaler* is a tunable parameter.

$$vrt(m_{j+1}) = vrt(m_j) + (i+1) \times scaler \tag{9.1}$$

The value of *scaler* is adjusted for different simulation experiments to increase or decrease the disparity of self event rates along the pipeline. With values of *scaler* greater than one, the timestamp increment given to self events at successive LPs in the pipeline increases and thus the chance of a rollback increases at LPs at the end of the pipeline.

9.1.5 Torus

In the **Torus** workload the LPs are laid out in a toroidal grid. Each LP only communicates with its North, East, West, South (NEWS) neighbours. The toroidal configuration results in the LP at one edge of the grid having a connection to the LP at the opposite side of the grid. Figure [9-4] shows a 4 $\times$ 4 torus (16 LPs) with the toroidal NEWS communication paths shown with solid lines to direct neighbours and dashed lines showing the communication paths formed by the toroidal wrap.

In the experiments reported in section 9.2.4 different sized toroidal grids are used. Fig-

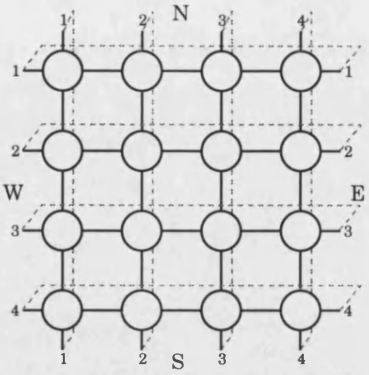

Figure 9-4: A **4** × **4** (**16** LP) **Torus** configuration.

ure [9-5] shows how the different grid sizes are constructed. For a grid size of **16**, block **a** is used; this is the same as is shown in figure [9-4]. For a grid size of **32**, block **b** is added to block **a**. Then for sizes **64**, **128**, **256**, and **512**, blocks **c**, **d**, **e** and **f** are added successively.

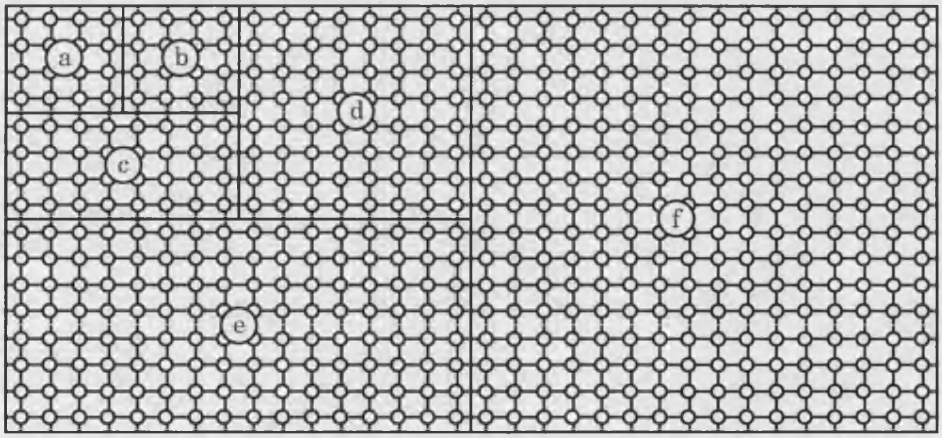

Figure 9-5: The **Torus** test models are constructed as follows: for a grid size of **16**, block **a** is used; this configuration was shown in figure [9-4]. For a grid size of **32**, block **b** is added and then for sizes **64**, **128**, **256**, and **512**, blocks **c**, **d**, **e** and **f** are added successively.

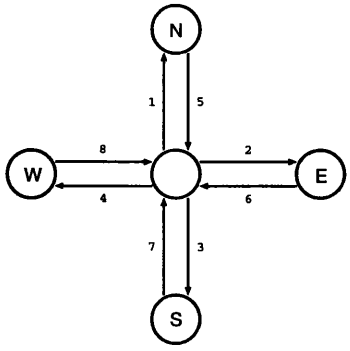

Figure 9-6: **Torus** workload message passing pattern for unit inter-arrival time testing of the BBS algorithm. A message is sent every 1.0 units of time and each message has a lifetime of 4.0 units of time. This results in one event occurring at each LP at each unit of virtual time.

Most of the tests presented in section 9.2.4 randomly select one of their NEWS neighbours to send a new event to on receiving an external event and use a timestamp increment taken from an exponential distribution. The exact messaging behaviour for each of these simulations will be described in section 9.2.4. Another test uses fixed time increments and the message passing pattern shown in figure [9-6]. Initially each LP sends an event to its N, E, S, W neighbours with timestamps 1.0, 2.0, 3.0 and 4.0 respectively. On receiving an event a new event is sent to the sender of the event just received with a timestamp increment of 4.0. This results in an arrival pattern where events arrive at each LP at unit time intervals, but with events from a particular neighbour appearing at eight time unit intervals. This provides an interesting test case for BBS since the probability of an inter-arrival time of 1.0 is 1.0, thus providing a simple way of assessing the effects of the adaptive parameter adjustment in BBS.

9.1.6 Pucks

The **Pucks** simulation model uses an implementation of the *distribution list algorithm* (DLA), Steinman and Wieland [67]. The simulation of moving objects creates dynamic changes in system load balance that can cause performance problems for parallel simulators, Lubachevsky [35].

The distribution list algorithm is very complex and will not be described in detail here. I refer the reader to Steinman and Wieland [67] for an in depth explanation of all the LP types and event types used. The following paragraphs give a basic description of what the algorithm involves as well as some of the parameters used within the DTW implementation.

The DLA uses three LP classes; two concrete and one abstract. The concrete classes represent grid objects, objects that represent a region of the modelled area in which entities move

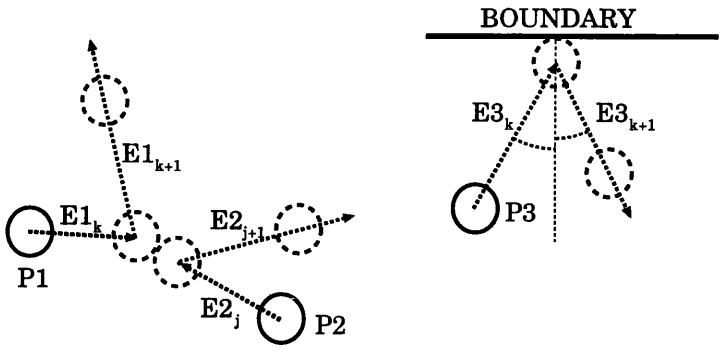

Figure 9-7: Pucks colliding into each other (left) and into the boundary of the grid area (right). All collisions are modelled as being perfectly elastic. In each case a collision causes a puck's existing EOM to be replaced by a new one (e.g., replace EOM $E1_k$ with EOM $E1_{k+1}$) which is then distributed by the local EOMMAN to any executive at which it is needed.

around, and equation of motion manager (EOMMAN) LPs that are used to hold any equation of motion (EOM) polynomial objects that may be required on the same executive as the EOMMAN resides. The abstract class represents moving objects and is specialised to particular types of movers; they are specialised to pucks in this case.

Each executive has at least one EOMMAN. The EOMMAN is unusual as it holds data (the EOMs) that it shares with mover objects on the same executive, with events being passed between the EOMMAN and the mover objects to inform a mover when an EOM is no longer valid. The intention of this is to save memory and save copying polynomial objects, since only one copy of any EOM is required on each executive. The DLA stipulates that there can be more than one EOMMAN per executive so that one EOMMAN does not have to handle all the EOMs that might be required at an executive. In the implementation in DTW, there is one EOMMAN per executive for each one hundred movers in the system. Though grid objects can handle arbitrary shaped areas, the grid object LPs in the **Pucks** simulator each model a square smooth surface.

The **Pucks** simulator works by puck LPs informing the LPs representing the grid squares their puck pass through, or pass near to, that they are in their vicinity. The grid square then informs the puck LP which other pucks are close to it and the puck then retrieves the EOMs of these "close" pucks from the local EOMMAN LP by exchanging event messages; pointers to the EOMs are passed to the puck LP from the EOMMAN. Then each puck LP can perform its own collision detection calculation to determine if a collision will occur and if so what its new EOM will be. When a new EOM is calculated it is passed to the local EOMMAN (or the local EOMMAN representing this mover in the case where more than one EOMMAN is

used per executive), which in turn distributes the EOM to any other EOMMAN that needs this new EOM. For a description of exactly how EOMs are distributed and how grid objects are informed of movement see Steinman and Wieland [67].

In each **Pucks** simulation the number of LPs involved can be calculated from equation (9.2), where K is the number of movers (pucks in this case) handled by each EOMMAN LP.

$$|LPs| = |pucks| + \text{grid\_width} \times \text{grid\_height} + |executives| \times K \tag{9.2}$$

Since there is one EOMMAN per executive for each one hundred movers in the DTW implementation of the distribution list algorithm, K is calculated from equation (9.3).

$$K = \left\lceil \frac{|movers|}{100} \right\rceil \tag{9.3}$$

In the simulations for which results are presented in section 9.2.5, each puck is a circular disk with diameter $\frac{1}{30}^{th}$ the width of a grid square. It should be noted that this is not the most efficient way of simulating pucks since any collision between a pair of pucks is calculated twice. The **Pucks** model is really only meant as a test model for the DLA implementation but does provide a useful benchmark due to the chaotic nature of the puck collisions.

9.2 Results

This section presents results for simulations using the models described in section 9.1. The main results reported relate to the wall clock time taken to run the simulations using the BBS algorithm and the more conventional LTF scheduling algorithm. The results were obtained by running the system on the MACI Alpha cluster at the University of Calgary. The Alpha cluster is a group of DEC personal workstations each with a single 500MHz Alpha EV56 CPU, connected by a Myrinet system area network.

9.2.1 BeRisky

Figure [9-8] shows the running time for simulations using the BeRisky model run on 8 executives (with one executive per PE) using both BBS and LTF scheduling. Results are shown for simulations using 1, 2, 4, 8, 16 and 32 LPs per executive. Despite being such a simple model, the results obtained show an interesting property of the BBS algorithm.

With 8 LPs (1 per executive), the simulator using BBS runs 0.8 times as fast as when LTF is used. This can be explained by the added overhead associated with BBS. With 1024 LPs (128 per executive), the BBS system runs 1.3 times faster than the system using LTF scheduling.

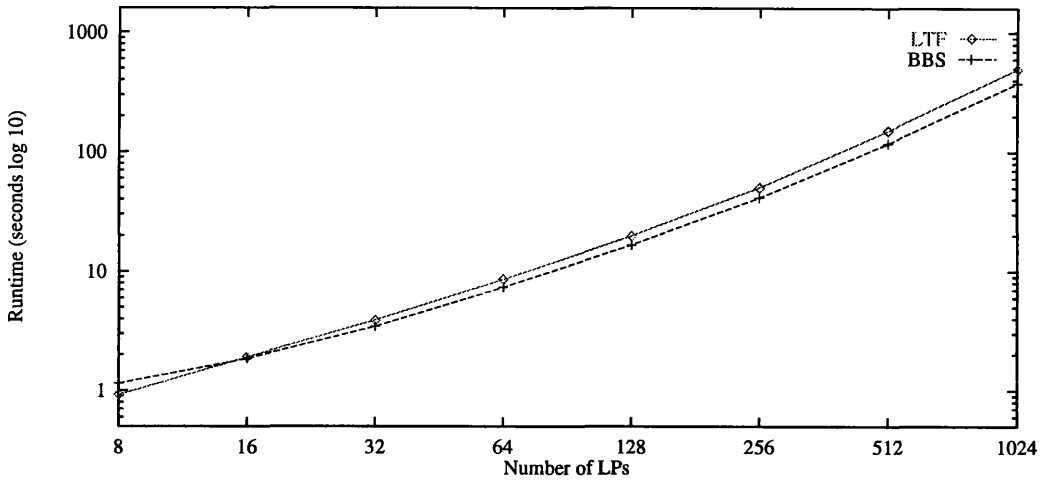

Figure 9-8: Running times of **BeRisky** simulations run on 8 executives. Simulations where run with 8, 16, 32, 64, 128 and 256 LPs, i.e., with 1, 2, 4, 8, 16 and 32 LPs per executive.

The result on 1024 LPs seems counter intuitive if BBS is considered merely as an execution throttling algorithm, but its performance can easily be explained by considering the effect of multi-event execution (MEE) (see section 7.4). Since no rollbacks occur, as is guaranteed by the construction of the **BeRisky** model, the maximum number of events executed in each LP execution session increases to MEE_max (see section 8.5.2). Also, the lack of rollbacks leads to the execution belief level decreasing to min_BBS_EBL (see section 8.4) at each LP. Together these two properties result in MEE_max events being executed during each LP execution session. The value of MEE_max is set to 5 in DTW. This improves the program cache locality and amortises the LP scheduling cost over MEE_max events. For the 8 LP case, since the same LP will be scheduled each time, the cache locality advantage of BBS over LTF is not seen. In this case, even though the scheduling cost is lower for BBS, the cost of performing the belief calculation leads to the slightly worse performance of BBS compared to LTF scheduling. This issue is discussed further in section 9.4.

9.2.2 Echo

Results are presented for simulations using two versions of the **Echo** workload (see section 9.1.3). In the first, the **Low Load Echo**, the only cost associated with receiving an event is the generation of a new event; no additional synthetic load is added. In the second, the **High Load Echo**, 30 integer values are updated each time an event is executed.

Low Load Echo

Figure [A-1] (in Appendix A) shows the results of simulations using the **Low Load Echo** model with two LPs, with each LP allocated to its own executive. Results are shown for simulations with R_2 event rates from 1.0 to 128.0 (R_1 is set to 1.0 in all cases). Three graphs are shown, with each plotting a set of values obtained from an experiment against the R_2 event rate used in the experiment. The top graph plots the amount of wall clock time taken to complete the simulation. The middle graph plots the total number of events rolled back during the simulation. Finally, the bottom graph plots the lengths, in terms of events rolled back, of the rollbacks that have occurred; both the maximum and average lengths are given.

From figure [A-1:top] it can be seen that as soon as the R_2 event rates get larger than 1.0 the echo failure mode starts to inhibit the performance of the system using LTF scheduling. In fact with an R_2 rate of 128.0 the BBS simulator runs 18.4 times faster than the LTF system. Notice from figure [A-1:middle] that the number of rollbacks that occur using the two scheduling policies is similar; the reason for the slowdown for the LTF scheduling system can be seen from figure [A-1:bottom]. Both the maximum and average rollback lengths with BBS stay stable as the R_2 event rate is adjusted between 1.0 and 128.0. With LTF scheduling, the maximum and average length values climb as the event rate increases from 1.0 to 8.0 and then stabilise with the maximum remaining at approximately twice the average in the rest of the simulation runs.

High Load Echo

Figure [A-2] shows the effect of adding load to the LPs involved in the echo with all other parameters as used for the experiments for which results are presented in figure [A-1]. The load is in the form of a loop in which elements are selected at random from a vector of automatically state saved integers and their values updated. In these examples, 30 values were updated each time an event is executed. As can be seen, BBS still performs better than LTF, but by a much smaller amount than in the low load case presented in figure [A-1]. This is explained by the smaller number of events that are executed at each LP between when the event message from the previous (in this case only other) LP arrives. Note that since DTW uses incremental state saving the cost of the rollbacks is also increased by adding the state updates to the low load model. This cost is present in both the BBS and LTF scheduling versions, but appears not to have unduly effected the LTF scheduling system in this particular case. In this case the BBS system only runs 1.16 times faster than the LTF scheduled system when the R_2 event rate is set to 128.0.

9.2.3 Pipeline

Figure [A-3] shows results for **Pipeline** simulations using 4 LPs on 4 executives with different values for the tunable parameter *scalar* used in function (9.1). Figure [A-3:middle] shows that the BBS system suffers a far greater number of rollbacks than the LTF scheduling system. Despite this the BBS system consistently outperforms the system using LTF scheduling with a runtime 7.8 times faster for a *scalar* setting of 1.0. A graph plotting the speedup achieved by BBS over the LTF scheduling system in this case is shown in figure [9-9]. Looking at fig-

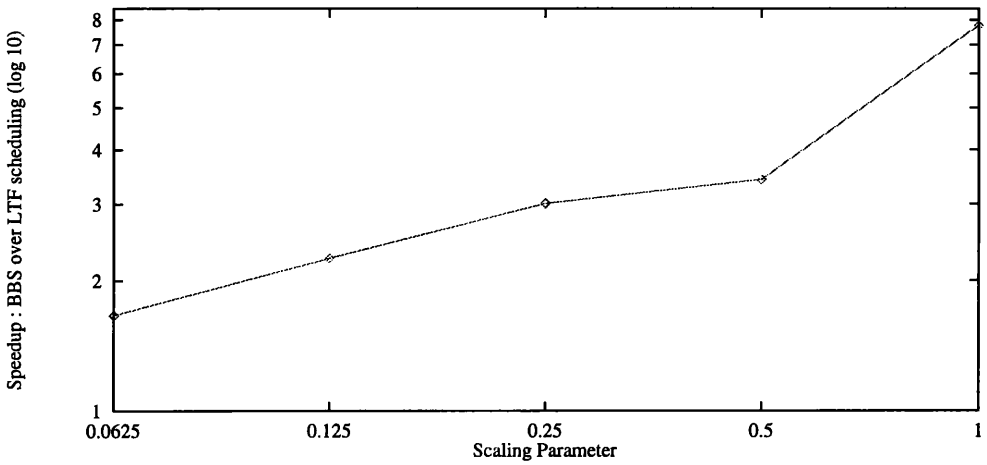

Figure 9-9: Performance increase using BBS rather than LTF scheduling for the Pipeline simulation experiments for which full results are presented figure [A-3] in Appendix A.

ure [A-3:bottom] we see that the maximum rollback lengths experienced by the LTF scheduling system are far greater than those occurring in the BBS system. These long rollbacks are likely caused by long rollback cascades rippling down the pipeline. The fact that the LTF scheduling system experiences less rollbacks is most likely explained by the long rollbacks at the end of the pipeline delaying the progress of these LPs enough that more events arrive in time not to cause a rollback.

Figure [A-4] shows results for simulations using the same parameters as those shown in figure [A-3] but with load applied. The load takes the form of 30 automatically state saved integer updates each time an event is executed. In this case BBS outperforms LTF scheduling but only by a very small amount. In this case the extra load on the LPs near the end of the pipeline prevents them from executing large numbers of events that would then need to be rolled back. This is similar to the effect seen in the **High Load Echo** model.

9.2.4 Torus

Simulations using three configurations of the **Torus** workload (see section 9.1.5) are presented in this section. The results for these simulations are shown in figures [A-5], [A-6] and [A-7] in Appendix A. Each of these simulations was run on 8 executives.

Figure [A-5] shows the results of **Torus** simulations in which one initial event is generated at each LP. On receiving any event an LP picks (using a uniform probability distribution), which of its NEWS neighbours to send a new event to. In this case all event timestamp increments were calculated from an exponential distribution with mean 1.0. It can be seen that in this case the performance of BBS and LTF scheduling are almost identical in all respects. It is not surprising that BBS does not show any advantage here since there will not usually be enough events available to execute at any LP to benefit from the effects of MEE.

Figure [A-6] shows results for simulations with the same parameters as used for the simulations results are presented for in [A-5], except that 4 initial events are generated at each LP. In this case BBS offers some advantage over LTF scheduling. At a grid size of 16, the BBS system is 1.5 times faster than the LTF scheduled system and at a grid size of 512 it is 1.2 times faster than the LTF scheduled system. From the middle and bottom diagrams it can be seen that the length of rollbacks is similar for both scheduling techniques, but slightly fewer rollbacks are observed from the BBS system. Here there are enough events in the system that the benefits of MEE start to show themselves.

Finally, figure [A-7] shows a **Torus** simulation using the fixed messaging pattern and timestamp increments shown in figure [9-6]. In this case, as expected BBS performs better than LTF scheduling, but not by as much as may be expected from this carefully constructed model. Indeed an early version of BBS performed much better on this test model, but performed poorly on many other tests. In this case the predictive property of the BBS algorithm should avoid all false blocking while never experiencing a rollback; this is exactly what happened when this simulation was tested with an early version of BBS. The addition of the adaptive variables that allows the BBS system to adapt to changing event arrival patterns has actually led to a performance reduction in this case, one of very few cases where this occurred. Despite this reduction in overall performance from the non-adaptive BBS, the current version of BBS still completes the 512 LP simulation 1.2 times faster than the LTF scheduling version.

9.2.5 Pucks

Figure [9-10] shows the performance of simulations using the **Pucks** model on a very large (25×25) grid. With this grid size, no rollbacks occurred during any of the tests; this can be explained by the lack of interaction between pucks on the large surface. As was the case with the **BeRisky** model, the BBS system beat the LTF scheduling system despite the lack

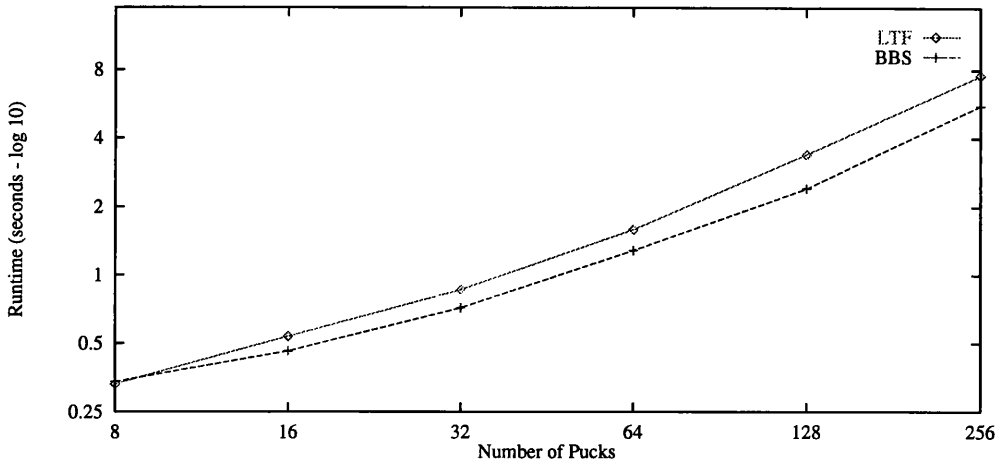

Figure 9-10: Runtimes of **Pucks** simulations run on a **25** $\times$ **25** grid.

of rollbacks. With 256 pucks the BBS system was 1.36 times faster than the LTF scheduled system. This can again be attributed to the better cache locality and amortised scheduling cost afforded by BBS. It was observed that an average of just under three events were executed during each LP execution session in all of the tests using the BBS system. In fact, once the BBS adaptive parameters settled, the puck LPs executed MEE_max events (5 in this case) per execution session, with other LPs such as the grid square LPs only having one or two events to execute in most sessions. The number of events executed by the puck LPs is explained by the generation of self events used to check their current location in case new grid squares need to be informed of their presence.

The results in figure [A-9] are for different numbers of pucks on a 15 $\times$ 15 grid. The results in figure [A-8] are for simulations using the same number of pucks as were used in the experiment run on a 15 $\times$ 15 grid, but this time running on a 5 $\times$ 5 grid. With these simulations the size of the grid does not change, so as the number of pucks increases the number of collisions that occur also tends to increase. This in turn increases the number of events exchanged between the pucks and the EOM managers and increases the chance that rollbacks will occur. Other than that the effect of increasing the number of pucks (and therefore LPs), is similar to the effect of increasing the number of LPs in the **SimLoad** models.

Unlike with some of the other models, the **Pucks** simulations showed good speedups even with small numbers of LPs. For the 4 puck simulation the BBS simulator ran 1.7 times faster than the LTF version with the larger grid and 2.03 times faster than the LTF version on the smaller grid. For simulation using 256 pucks, the BBS system ran 1.73 times faster than the LTF scheduling system using the larger grid and 1.85 times faster than the LTF scheduling system using the smaller grid. The speedup achieved with BBS over the LTF scheduling scheme

with the 5 × 5 grid size are shown in figure [9-11].

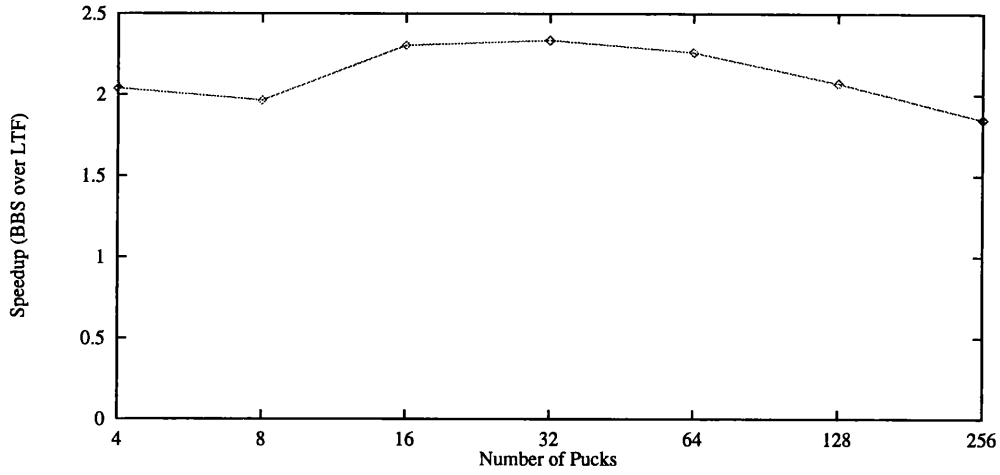

Figure 9-11: Graph showing the speedup gained by using BBS rather than LTF scheduling on an 8 executive **Pucks** simulation with a 5 × 5 grid size. For full results for these simulations see figure [A-9] in Appendix A.

This success with the **Pucks** simulator is interesting in several ways. For one thing, the **Pucks** model was not used in the development of BBS. Only the **SimLoad** models were used in the development process, so BBS wasn't tuned to perform well with the **Pucks** simulation model. Another point is that the **Pucks** model had been extensively tested with event execution throttling algorithms including an implementation of the throttled Time Warp algorithm, Reiher and Jefferson [47], and an implementation of the Elastic Time Algorithm[2], Srinivasan and Reynolds [60], with no advantage being seen over the LTF scheduling system. With both these algorithms large performance penalties were observed when the throttling parameters were set too conservatively. It appears that it is the MEE capability of BBS that makes it perform well with this simulation.

9.3 Saved Acknowledgements

The main purpose of this chapter is to consider the effects of the BBS algorithm on system behaviour. This section departs from this slightly to present results collected during the test simulations on the number of acknowledgement messages required to calculate GVT. As described in chapter 5, DTW uses a lazy message acknowledgement scheme to attempt to reduce the number of control messages required to calculate regular GVT updates in a distributed mem-

[2]Note that in order to make ETA work effectively in DTW, SBT estimation had to be used since NPSI GVT updates were not available.

ory environment. This is done by using a sequence numbering scheme to allow one acknowledgement to act as an acknowledgement for a series of messages. Also, acknowledgements are piggy-backed onto messages to remove the need for some of the explicit acknowledgements to be sent.

Simulation		Figure	Messages	Acks	Acks Saved (%)
Echo	($R_2 = 128.0$)	[A-2]	1746	599	65.7
Pipeline	($scaler = 1.0$)	[A-3]	934	146	84.4
Torus	(512 LPs)	[A-5]	120235	2326	98.1
Torus	(512 LPs)	[A-6]	79318	93	99.9
Pucks	(256 pucks)	[A-8]	8644	67	99.2

Table 9.1: Statistics for the number of acknowledgement messages that had to be sent in selected simulations. The *Figure* column indicates the figure in which performance results are reported for the simulation. The *Acks Saved* column shows the percentage of acknowledgements that have been saved over a system that acknowledges every message.

Table [9.1] shows the number of acknowledgement messages that were sent in each or the simulations for which results are presented. Also shown are the number of simulation messages sent via the message passing layer and the percentage of simulation messages for which no acknowledement message had to be sent.

In the 2-LP (1 LP per executive) **Echo** simulation [A-2], the pair of directed edges in the communication graph between the two LPs share a single inter-executive bi-directional communication channel. For each message sent from LP_A to LP_B a message is returned from LP_B to LP_A. This message acts as an implicit acknowledgement for the last message sent from LP_A to LP_B and vice versa. Therefore it seems likely that very few acknowledgements will be sent. As can be seen, in fact one third of the messages are acknowledged, which probably means that most of the messages that arrive when the GVT calculation is in **mode 2** have timestamps less than the receiving executives EVT and therefore are explicitly acknowledged.

With the **Pipeline** model, for all LP pairs LP_i, LP_{i+1}, messages are passed along the channel (**i,i+1**), from LP_i to LP_{i+1}, but not along (**i+1,i**). Therefore for the 8 LP test (1 LP per executive) [A-3], no messages are passed back along the bi-directional inter-executive channels, so no implicit acknowledgements are made. In this case the saved acknowledgements recorded are solely due to using a batched message acknowledgement when required rather than sending an individual acknowledgement for each message received.

In the **Torus** model all communication links between LPs are treated as bi-directional, so many messages are acknowledged implicitly by messages being sent back along the links. In the model in which only one initial event is generated per LP, 1.9% of messages needed

an explicit acknowledgement. In the model where four initial events are generated at each LP, 0.1% of messages needed an explicit acknowledgement. The improvement in this model can be explained by the greater message density increasing the number of implicit acknowledgements taking place.

In **Pucks**, the EOM manager LPs communicate with each other to communicate changes in each puck's equation of motion. This generates regular inter-executive communication which results in many messages being acknowledged implicitly. In the model for which results are given [A-8], only 0.8% of inter-executive messages needed explicit acknowledgement.

9.4 Discussion

The results presented here were collected during benchmarking runs on the final BBS implementation. The test models shown, along with some other SimLoad configurations and a game of life model were used for continual testing and development of BBS as well as for appraising other scheduling algorithms. The use of these models greatly effected the development of BBS. For example, the original version of BBS did not bound the number of events that could be executed during a single execution session. The use of the models led to the realisation that this could cause problems if the belief calculation was poor for any reason and the adaptive bound variable (MEE_max) was introduced. Also, early versions of BBS did not have SBT estimation. While this worked reasonably in some cases, all too often the system performed poorly due to executives waiting until the next GVT value arrived to execute events that were safe to execute but had low belief levels. This in turn slowed the progress of GVT, thus making the situation worse.

The version of BBS presented is a compromise between using belief information and adaptively trying to optimise the decisions made using the belief calculations. The adaption is based on how successful the decisions made have been in the recent past and how close a candidate event's timestamp is to a local estimate of the SBT.

The observation of how much benefit BBS was receiving from MEE led to some further experiments with MEE in Time Warp without the belief calculations. This system, which I will refer to as TW-MEE, showed a further improvement over BBS on the 1024 LP **BeRisky** simulation. With a static window of 5 events per scheduling session, TW-MEE ran 1.2 times faster than BBS and 1.5 times faster than LTF. Also experiments with BBS but without MEE (only one event executed per session) BBS ran 0.9 times the speed of LTF in the 1024 LP **BeRisky** simulation.

The results with TW-MEE might suggest that the use of MEE without the added belief calculations in BBS might provide a good algorithm. Some further experiments with more realistic models (Torus and Pucks) showed that TW-MEE did not in fact perform well in these

cases. Initially I believed this problem to be due to the lack of adaptiveness in the selection of the event execution count value, but installing the adaptive scheme from BBS (see section 8.5.2) did not solve the problem. The evidence suggests that while the maximum event count in BBS improves efficiency and reduces problems caused by inaccurate belief calculations, it is for the most part the rôle of the belief calculation preventing too many events being executed early that makes MEE in BBS effective.

9.5 Summary

This chapter presented the simulation models used to test DTW along with some performance results and analysis of these results. The results show that while the overhead associated with the belief calculations in BBS can have an impact on performance, these costs are often less than the combined costs of over optimistic behaviour and poor cache locality associated with LTF scheduling. The results obtained are better than expected since the primary rôle for which BBS was designed is to eliminate the worst case behaviour that can effect certain simulations using Time Warp.

All simulations with large numbers of LPs and events performed better with BBS than with LTF scheduling. It is clear that the use of multi-event execution (MEE) is the main factor in this performance increase. It is also apparent that using MEE without dynamic control of the number events executed in an LP execution session was not in general a good option, as it often resulted in large numbers of additional rollbacks and a performance penalty. The dynamic control afforded by BBS allows MEE to be used effectively.

Also presented were statistics on the number of acknowledgement messages required to calculate GVT correctly. These showed that for the simulation experiments presented between 65.7% and 99.9% of acknowledgements that would have been required with full message acknowledgement were not required using the lazy message acknowledgement scheme implemented in DTW.

Chapter 10

Conclusions

The purpose of this thesis was to investigate the usability of PDES on distributed memory parallel computers. This class of computers includes monolithic massively parallel processors (MPPs) as well as networks of workstations or small compute servers, connected via a high speed network. In order to carry out this investigation, a Time Warp system called *distributed time warp* (DTW) has been developed.

One part of the work was to perform an investigation into the feasibility of a transparent state saving system. While the results were not entirely satisfactory, the work gave insight into the problems involved in trying to achieve transparent state saving in a general purpose programming language. Some of these insights are described in section 10.3. The largest part of the work considered how event scheduling policies effect the performance of a distributed memory Time Warp system and how adaptive calculations could be used to achieve good performance without the user needing to tune the scheduler for their simulation. It was found that reasonably accurate information about the state of the system as a whole was required which led to work in the area of GVT calculation; this work is summarised in section 10.4. Then the scheduling algorithm, *Belief Based Scheduling* (BBS), that was developed is described in section 10.5. Finally, closing remarks are given in section 10.6.

10.1 Why Time Warp?

The main algorithms used for PDES are Time Warp and algorithms based on the conservative Chandy-Misra-Bryant (CMB) algorithm. These are not the only algorithms that have been proposed, with ANR algorithms and synchronous conservative algorithms also being available. The synchronous conservative algorithms are unlikely to perform well in a distributed memory environment for simulations with low lookahead since the cost of communication in a distributed memory environment is generally far larger than the cost of computation, making

synchronisation too expensive. ANR algorithms are of interest for interactive simulation environments, but have been shown to be less effective than other classes of algorithms in the batch processing environment in which most parallel simulation is performed.

This leaves the Time Warp and CMB based algorithms. One thing that has become increasingly clear throughout the period of this thesis work, is that it is going to be extremely difficult to make Time Warp as effective at speeding up some classes of simulation models as optimised algorithms based on the CMB channel synchronisation technique.

Where Time Warp wins is that it is capable of simulating systems with zero lookahead cycles in the communication graph defining the interactions between physical processes. This can lead to a major advantage in the speed in which models can be constructed since often, while it is possible to find ways of representing a physical system such that every cycle in the communication graph has lookahead, it is easier to model the system with some zero lookahead cycles.

For a general distributed memory PDES system Time Warp still appears the best option at the moment. A bigger question has to be whether a general system makes sense at all. Since CMB based algorithms such as *Critical Channel Traversing*, Xiao *et al.* [75], have shown themselves to be so effective, it seems that any system that suits the channel based modelling methodology should use a CMB based algorithm. Therefore, the "general" PDES system should only be used for simulations that cannot be run using a CMB based system. It is possible that the best general solution will be to allow conservative and optimistic components to interact in the same simulator. This would require the modeller to provide as much topology and lookahead information to the system as they have, even if in some cases the information is that a physical process has no lookahead and could communicate with a very large number of the other physical processes in the system. Such information could be used to partition the simulation into conservative and optimistic components that can interact without the chance of false messages crossing from an optimistic component to a conservative one. There is still much research to be done in this area.

10.2 Distributed Memory Issues

This thesis work has concentrated on PDES and more particularly on Time Warp in a distributed memory programming environment. Using a distributed memory programming model is attractive since it allows for a wide range of parallel computers, including low cost cluster computers to be used. Using a distributed programming model does have some disadvantages however. For one thing, event messages have to be packed and unpacked in order to have them transferred from one processor to another. It is also likely that the cost of transferring an event message from one executive to another will be greater on a distributed memory computer than

on a shared memory computer, though modern shared memory computers have complex hierarchical memory systems that can also introduce high latencies. The greater the time taken for an event to reach its destination, the greater the chance that it will arrive late leading to rollback. Possibly the biggest problem is the difficulty of implementing a dynamic load balancing scheme in a distributed memory environment.

Load balancing is very difficult for a general Time Warp system on a distributed memory computer. This is an issue that was addressed with some degree of success within TWOS, Reiher and Jefferson [47], but TWOS places restrictions on the construction of LPs in order to achieve this. In TWOS, the whole state of an LP has to appear as a contiguous piece of memory. Pointers were allowed, but only if they point back to an address within the LP's contiguous memory block. This enables an LP to be migrated by copying the whole of the block of memory to another executive. This places restrictions on the use of dynamic memory that I wanted to avoid.

The packing and unpacking of LPs that is required for distributed memory LP migration could be achieved using the parameterised state classes used for state saving in DTW; indeed this was attempted in DTW. This clearly suffers from the same problems associated with type coercion in C++ that affects the state saving system and is therefore not a robust solution to the problem. Again, this would be far better solved using compiler support.

Another problem faced by a dynamic load balancing scheme for a distributed memory Time Warp system is *when* LPs should be migrated. If an LP is migrated with an LVT greater than GVT, then old state and events will have to be migrated with the LP. On the other hand, if LPs are migrated only when GVT reaches their LVT, no old state needs to be migrated.

In shared memory the problem of packing and unpacking of LPs for migration is eliminated since the whole address space is accessible to every processor, so migration simply involves passing a single pointer from one executive to another. Also there is no need to move LPs at GVT since there is no extra cost associated with the LP holding old state when it is migrated. Therefore this is one area where shared memory Time Warp implementations have a great advantage over distributed memory implementations.

10.3 Handling State

In the early part of this thesis work, much time was spent considering the handling of LP state. My work in this area came to an end when Fabian Gomes published his thesis [26] that answered many questions that I had. Despite this the work I did has value for three reasons:

- I spent more time considering how state saving could be implemented transparently in the general purpose programming language C++ and gained insight into the problems involved in such an endeavour.

- I devised a system that allowed lists of data structures to be state saved such that list ordering was preserved.

- I implemented an automatic and mostly transparent state saving and state recovery system that simplified the construction of simulation models used for testing.

It became clear that the implementation of a transparent state saving system in a general purpose computer language is fraught with difficulties. One problem is that the implementation of such a scheme relies on the users of the system using the data types in the way that was intended. DTW is written in C++, so the obvious way of implementing transparent state saving is by using parameterised types, or *templates* as they are called in C++. Unfortunately, C++ allows aliasing of addresses and simple coercion of types, so however careful the implementor of a system is in overriding built in operators to perform an action such as state saving, a user can inadvertently cause the wrong operators to be called.

Also, the question of when to use incremental state saving and when to use copy state saving is not clear cut. To transparently implement copy state saving in a general purpose language would be very difficult, which is in part what led to the adoption of incremental state saving. It is also difficult to avoid some of the performance problems that can be encountered with incremental state saving with a parameterised type based implementation. The conclusions of this work are the following:

- An automatic incremental state saving scheme can be implemented in a general purpose "object oriented" language such as C++.

- Incremental state saving does not always offer the best state saving solution, so relying on it as the only form of state saving in a general purpose Time Warp system will limit the system's usefulness.

- A compiler based solution is required in order provide efficient state saving that is fully hidden from the modeller.

The last point needs a slight addendum. While it is true that compiler based state saving is required to hide state saving from a modeller who has to program, this does not mean that it is required to satisfy all modellers. Many modellers never see the underlying code of the system and only plug together components supplied by application builders. This type of user will not be aware of state saving and so the way it is implemented will not matter to them.

My work in the area of state saving came to a halt since I did not believe that it was possible to provide a robust and efficient transparent state saving solution in C++ without compiler support. Providing compiler support is difficult for a language that is constantly evolving. This is not to say that I think that there is no more work to be done in the area of state saving. In

particular, ways of eliminating unnecessary state saves, ways of utilising reverse computation techniques and ways of mixing incremental and copy state saving in the same system need further examination.

10.4 Calculating GVT

The creation of an efficient GVT algorithm was not a goal when starting work with DTW, but proved to be an important part of the work. Many of the distributed memory GVT calculation algorithms available in the literature would not provide GVT information at the regular intervals that the event scheduling algorithms I wished to investigate required. This work started by implementing the pGVT algorithm, D'Souza *et al.* [15], an algorithm that purported to provide information at the regular intervals I required. Unfortunately, pGVT proved difficult to tune and often led to poor simulator performance when used with an execution throttling scheme. This, plus Phil Wilsey's suggestion that the number of control messages required to calculate GVT could be reduced using the sequence numbering technique of Lin and Lazowska [33], led to the distributed GVT calculation scheme presented in chapter 5.

The algorithm created uses GVT rounds; something that pGVT does not do and was initially avoided since it requires a message to be sent to the GVT manager from each executive during each round. While the fully asynchronous scheme used in pGVT was attractive, I found too many cases where large numbers of additional messages were required to correct high reported time values.

The term *system base time* (SBT) has been used in this thesis to differentiate between the lowest active timestamp in the system, referred to as the SBT, and the calculated lower bound on this value, the GVT.

10.5 Scheduling: Simplicity isn't Everything

The major part of this thesis work has concentrated on how events should be scheduled on each Time Warp executive. The event scheduling decision is central to the operation of a Time Warp system, since it is the decision of which event to execute at any time that governs how many rollbacks occur and more importantly, how quickly the simulation proceeds.

Many scheduling schemes were considered before the *belief based scheduling* (BBS) algorithm was created. Initially, BBS based its event execution decisions purely on forecasting future event VRTs by consideration of event VRTs observed at the scheduled LP in the recent past. Although this technique worked well in a few select cases, in general simulations run using this algorithm performed poorly.

The GVT calculation proved to be a major issue in the belief calculations. The greater the

latency in receiving new GVT calculations the more likely that poor belief decisions would be made. Much work was done considering how GVT could be calculated more quickly, but very regular GVT updates resulted in losing performance simply due to the increased inter-processor communication required. The final solution to this problem was to estimate the SBT locally at each executive. While this carried with it the risk of over estimation which could result in overly optimistic behaviour from the executive, in general it appears to overcome the problems associated with using GVT in the belief calculation.

The most interesting result from the work on BBS, is that even though the belief calculations add to the execution cost of most events, DTW using BBS was still able to outperform DTW using LTF scheduling in a large number of cases. In models where the "failure modes" were likely to occur, the performance of DTW using BBS was far superior to that of DTW using LTF scheduling. In all test simulations with very large numbers of LPs and events, DTW with BBS did at least as well as DTW with LTF scheduling. Only in simulations with small numbers of LPs and events were the additional costs associated with the belief calculations seen to impact on the performance.

The improved performance in simulations susceptible to "failure modes" is due to BBS preventing thrashing due to some LPs continually advancing their LVT values far into the future leading to long, and therefore time consuming rollbacks. The improvement observed in the general performance of simulations with large numbers of LPs was less expected, but is possible to explain from statistics collected during simulation runs. The advantage of LPs executing more than one event in each execution session, i.e., the advantage of *multi-event execution* (MEE), was clear. How much this is due to improved caching behaviour and how much due to reduced scheduling overhead is difficult to determine absolutely. It was also clear that simply executing more than one event per execution session was not in general the way to get good performance. There is a need for a dynamic selection of how far into the future events should be executed by an LP, and while it is not clear that BBS is the best way of achieving this, it has proved effective in many of the test simulations.

10.6 Closing Remarks

PDES in either a distributed memory or a shared memory programming environment is not a solved problem. While improvements are being made to both optimistic and CMB based algorithms, still no one algorithm is suitable for all simulation models.

This thesis has offered insight into the problems of implementing a Time Warp system in a general purpose computer language. The conclusion is that compiler support is required in order to provide an efficient Time Warp system that is usable by the general simulation community.

The main part of this thesis work has shown that it is possible to use statistical forecasting to eliminate the occurrence of "failure modes" in general models without imposing too large a performance penalty for models that do not suffer from these inefficient modes of behaviour using the standard LTF scheduling technique. While the benefits of statistical forecasting has been shown before, particularly in the work of Ferscha, this has been for particular systems for which there is some understanding of the arrival behaviour of events. This work has shown that just using simple statistical forecasting can lead to problems if the arrival behaviour is not understood. It has also shown that if heuristics are used to adjust the level of belief in the results of the forecasting and by taking advantage of the improved memory locality and amortised scheduling costs gained from the multi-event execution policy that such a scheme allows, an effective general scheduling algorithm is achievable.

Appendix A

Test Results

Results for the simulation experiments shown in Table [A.1] are presented in this Appendix. The experiments are described in chapter 9.

Figure	Simulation	Parameters
[A-1]	Pucks (low load)	R_2 values from 1.0 to 128.0
[A-2]	Pucks (high load)	R_2 values from 1.0 to 128.0
[A-3]	Pipeline (low load)	scalar values from 0.0625 to 1.0
[A-4]	Pipeline (high load)	scalar values from 0.0625 to 1.0
[A-5]	Torus (1 initial event)	grid sizes from 16 to 512
[A-6]	Torus (4 initial events)	grid sizes from 16 to 512
[A-7]	Torus (fixed messaging)	grid sizes from 16 to 512
[A-8]	Pucks (5 $\times$ 5 grid)	number of pucks from 4 to 256
[A-9]	Pucks (15 $\times$ 15 grid)	number of pucks from 4 to 256

Table A.1: Experiment results in Appendix A.

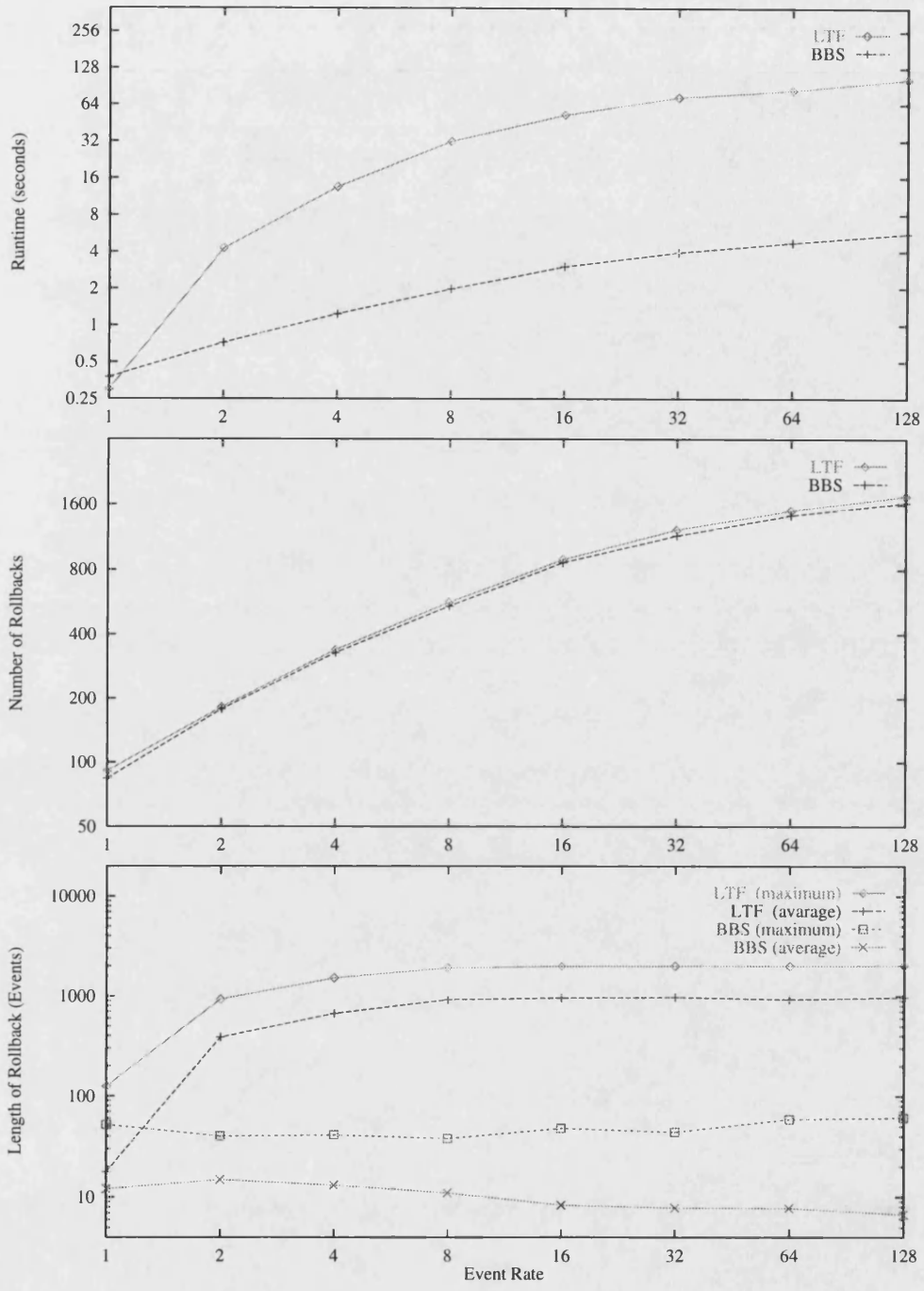

Figure A-1: **Echo** simulations with no additional execution load. With an R_2 event rate of 128.0, the BBS system completed the simulation 18.4 times faster than the LTF scheduled system.

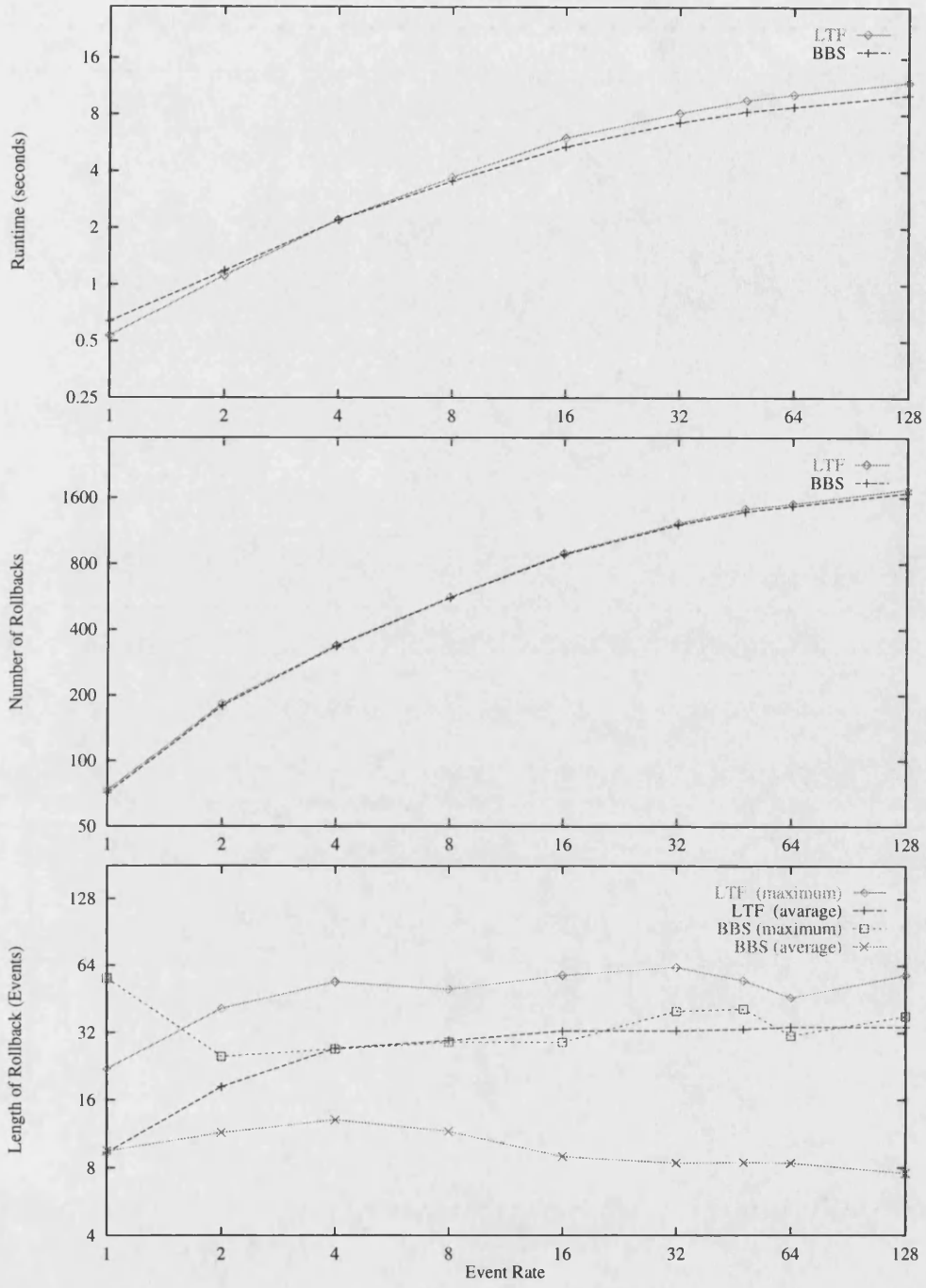

Figure A-2: **Echo** simulations with additional execution load applied at every event execution. In this case the speed advantage gained by using BBS is very small; just 1.16 times faster with an R_2 event rate of 128.0.

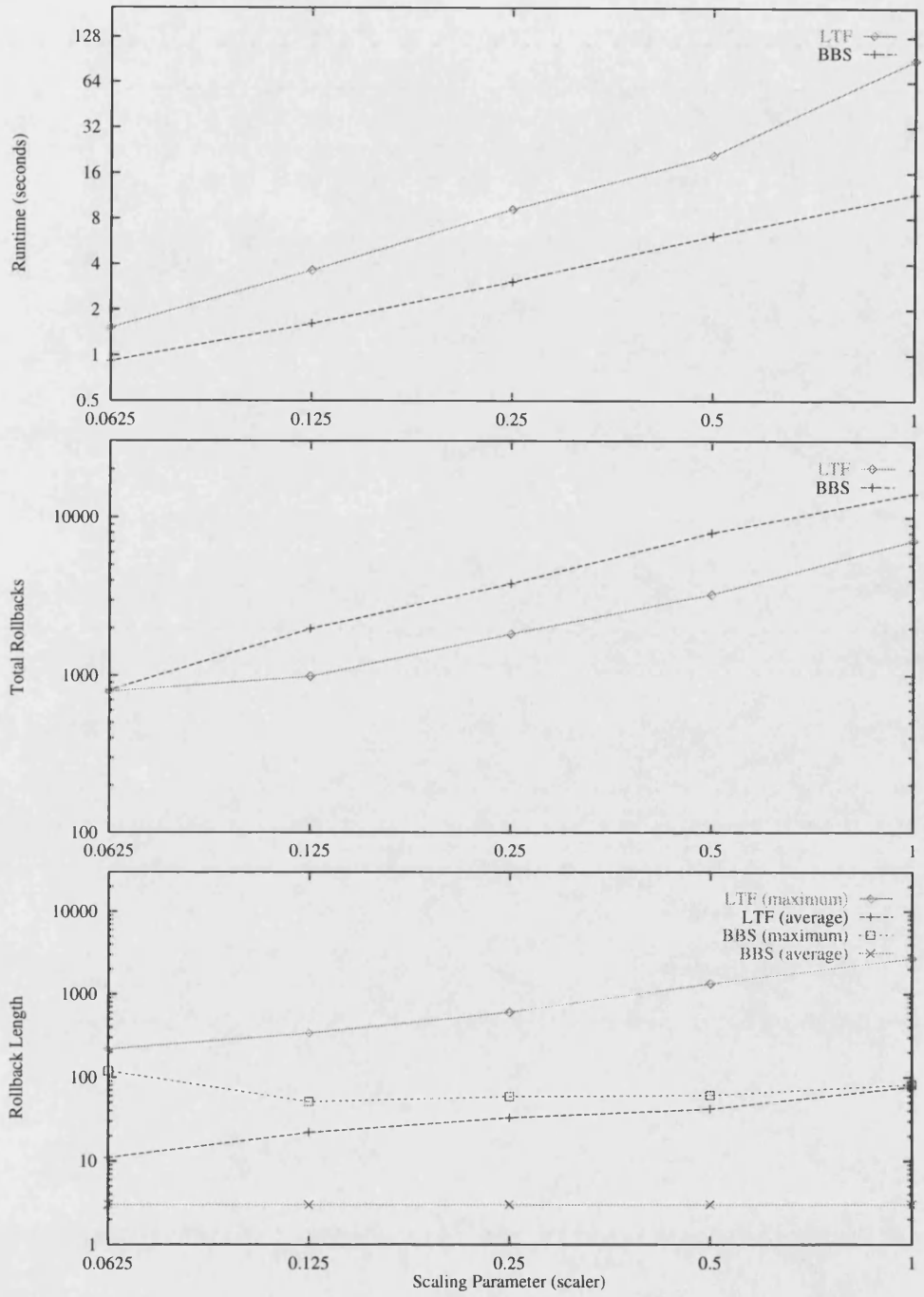

Figure A-3: Simulations running the **Pipeline** model with no extra load applied to event executions. Here the BBS system completed the simulation with *scalar* set to 1.0, 7.77 times faster than the LTF scheduled system. Notice that in this case the performance increase is due to the reduced rollback lengths experienced.

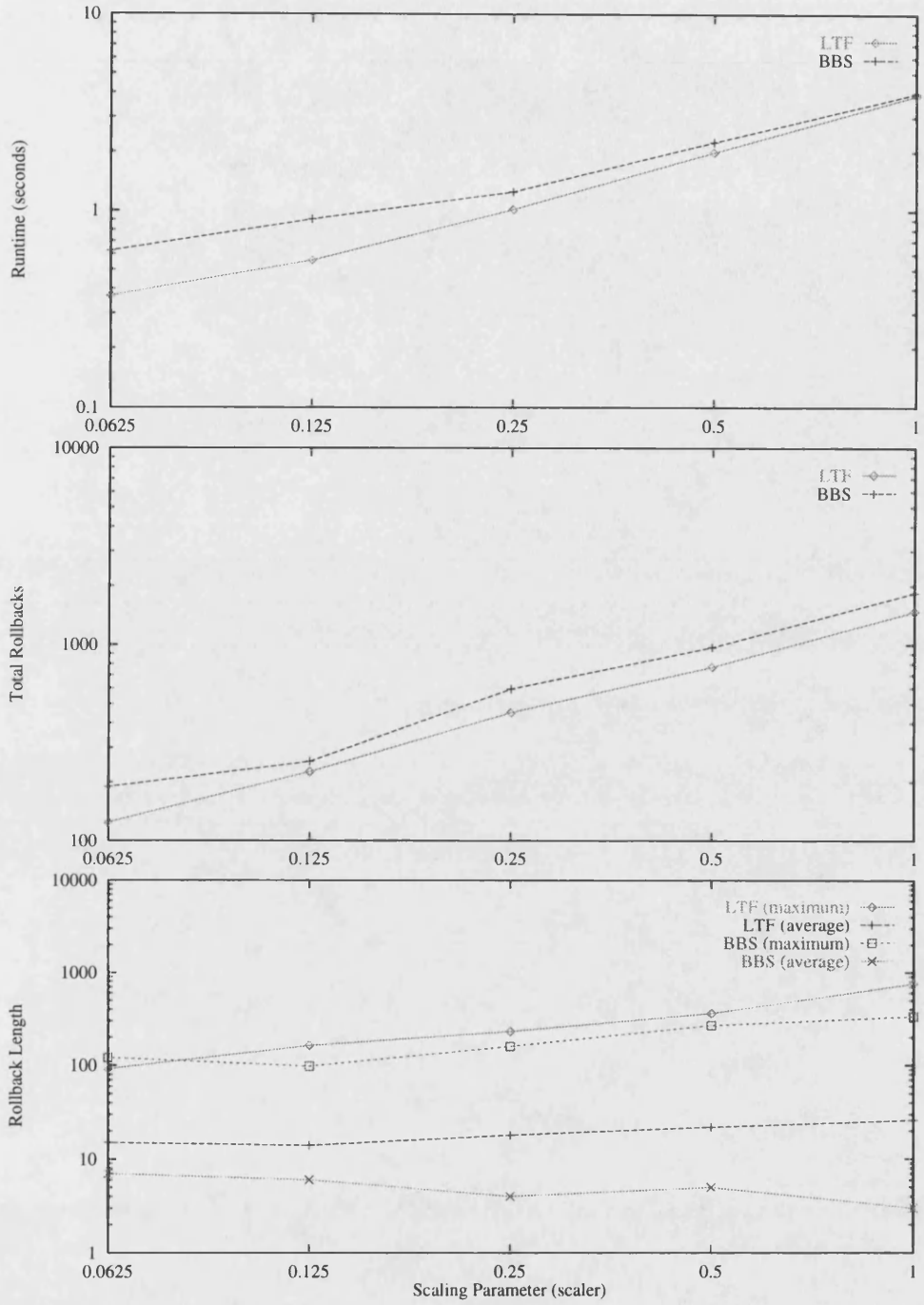

Figure A-4: Simulations using the **Pipeline** model with load applied during the event executions. In this case the performance advantage seen for BBS without additional load is gone.

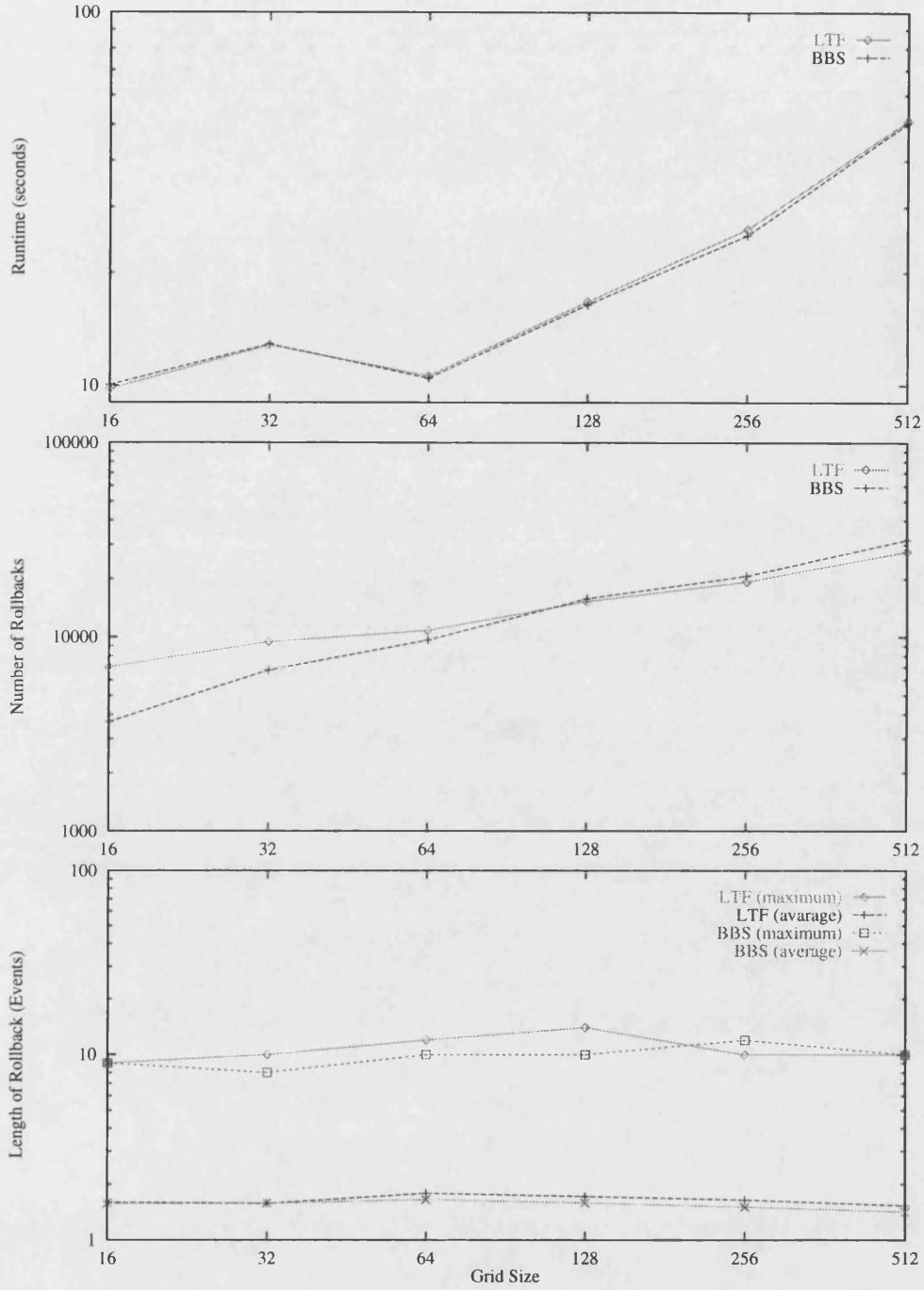

Figure A-5: Simulations using the **Torus** model with one initial event at each LP. Here the two systems work almost identically. This is thought to be due to most LPs only having one event to execute each time they are scheduled. This eliminates any advantage that could be gained from multi-event execution.

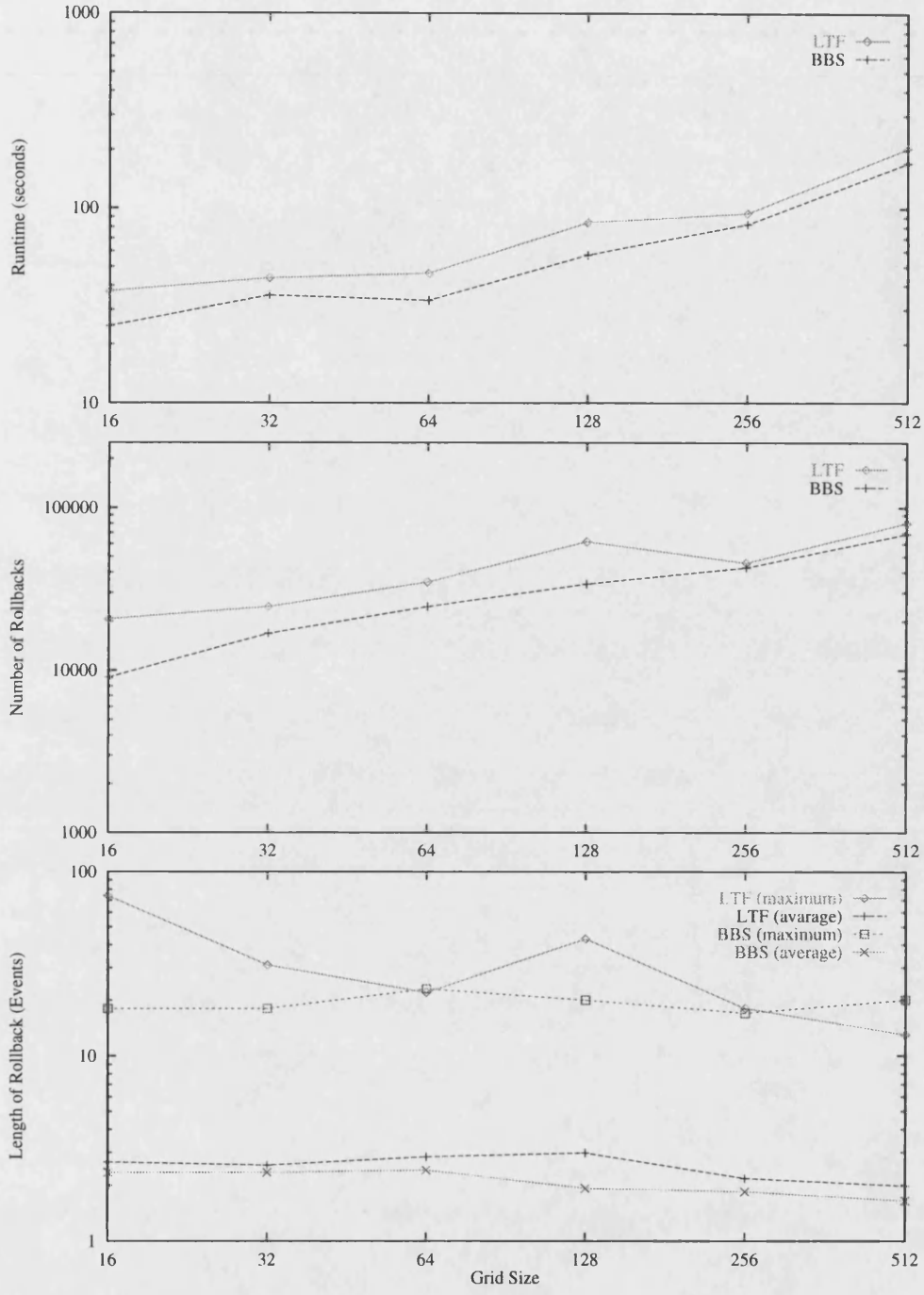

Figure A-6: Simulations using the **Torus** simulation model with four initial events at each LP. In this case the BBS system has a slight advantage, which is most likely due to the benefits of multi-event execution.

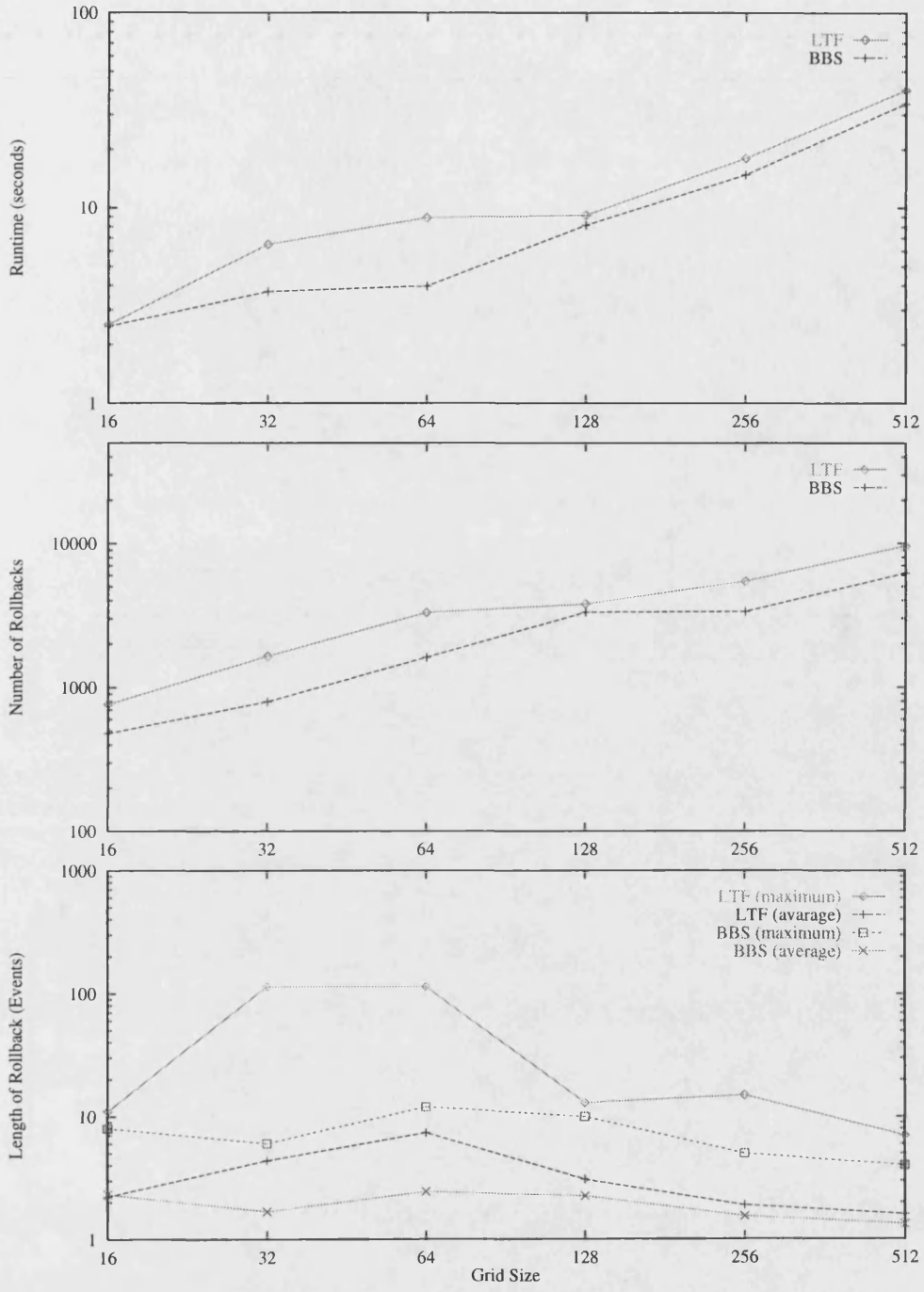

Figure A-7: Simulations using the **Torus** workload, using the symmetric message passing rules outlined in figure [9-6] in chapter 9. The BBS system is up to 2.24 times faster than the system using LTF scheduling, though this is reduced to 1.17 times faster at a grid size of 512.

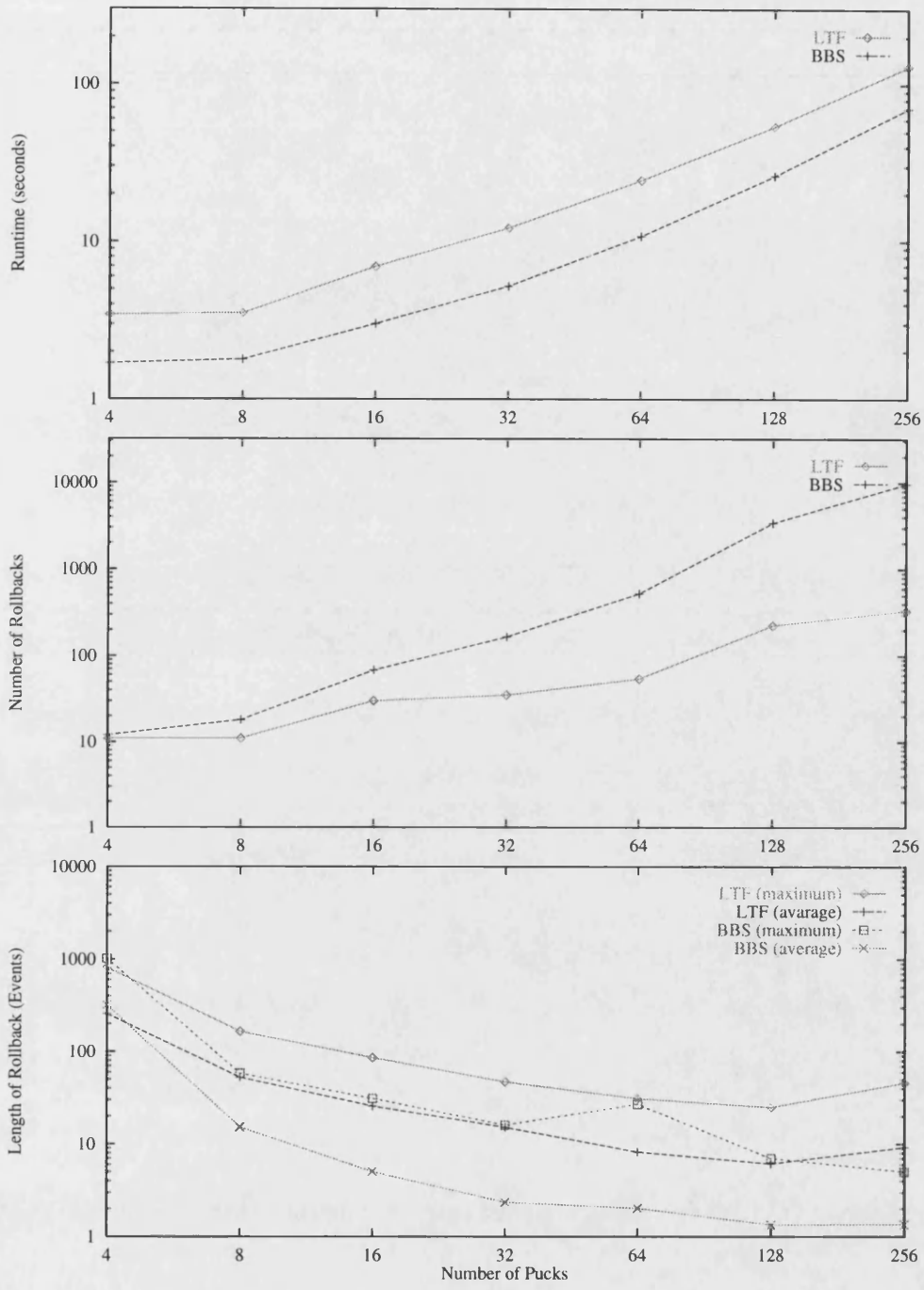

Figure A-8: **Pucks** simulation with different numbers of pucks on a 5×5 grid. Notice that some of the rollbacks that occurred were very long, and that BBS did not prevent this when there were few rollbacks. BBS adapts to prevent long rollbacks when many rollbacks occur.

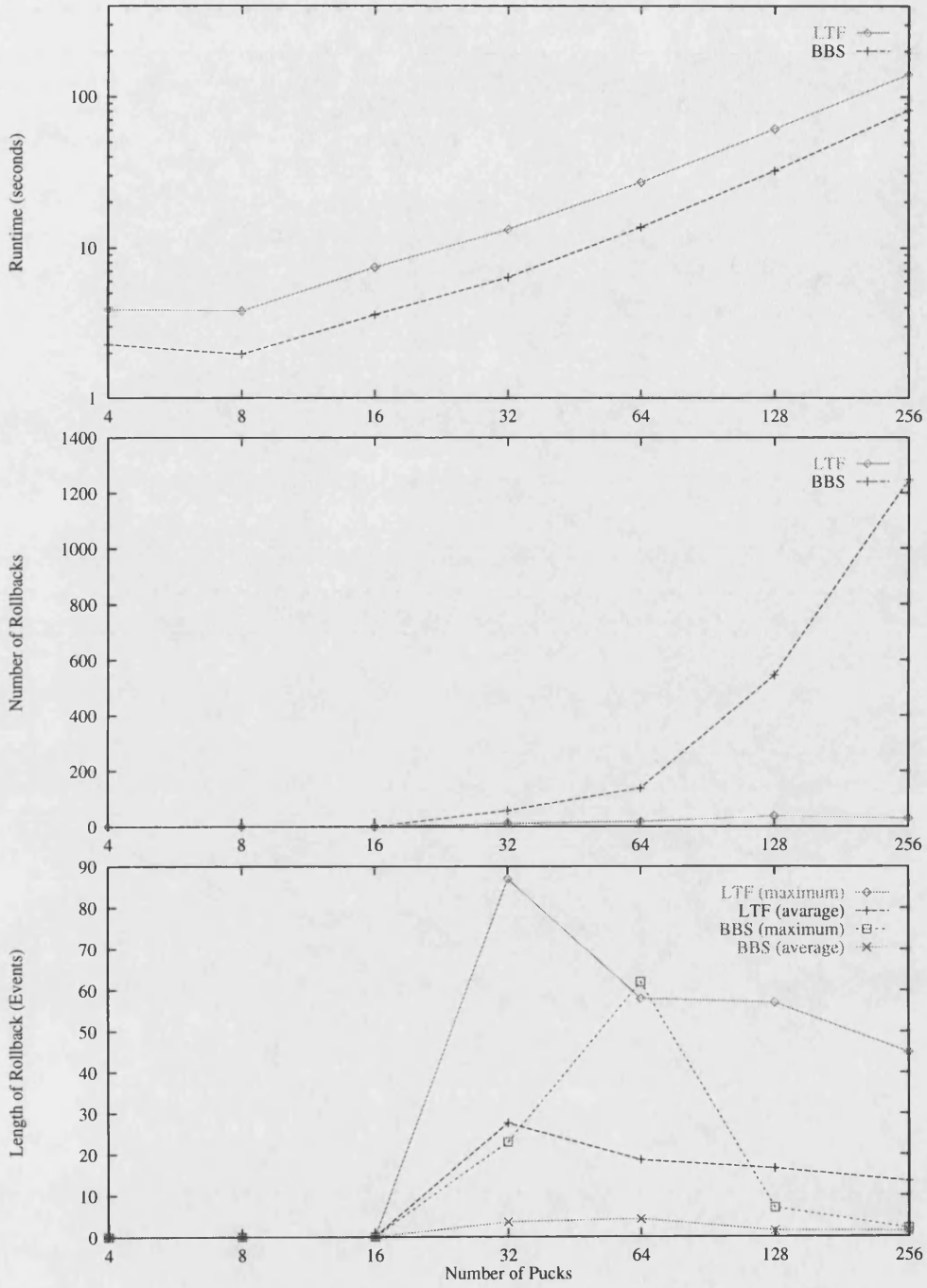

Figure A-9: **Pucks** simulations running on a $\mathbf{15} \times \mathbf{15}$ grid. In this case no rollbacks were observed with small numbers of pucks simply due to the lack of collisions being modelled.

Appendix B

Glossary of Acronyms

AAWP Asynchronous Adaptive Waiting Protocol (section 6.1.4).

ALB Asynchronous Lower Bound (section 5.1).

ANR Aggressive No Risk (section 1.1.1).

BBS Belief Based Scheduling (section 7).

CDF Cumulative Distribution Function (section 7.3.1).

CMB Chandy-Misra-Bryant (section 1.1.2).

CSS Copy State Saving (section 3.1.1).

DLA Distribution List Algorithm (section 9.1.6).

DTW Distributed Time Warp (section 2).

EMDO Event Message Delivery Object (see section 2.2).

EOM Equation of Motion (section 9.1.6).

EVT Executive Virtual Time (section 2.1).

GVT Global Virtual Time (section 2.1).

IAT Inter-Arrival Time (section 4.5).

ISS Incremental State Saving (section 3.1.2).

LMT Local Minimum Time (section 5.1).

LP Logical Process (section 1.1).

LTF Lowest Timestamp First (section 2.2.2).

LVT Local Virtual Time (section 2.1).

LQT Local Queue Time (section 2.1).

MPP Massively Parallel Processor (section 2).

MTL Message Transport Layer (section 2.2).

PDES Parallel Discrete Event Simulation (section 1.1).

PDF Probability Distribution Function (section 7.3.1).

PE Processor Element (section 2.2).

PP Physical Process (section 1.1).

RT Reported Time (section 5.1).

SBT System Base Time (section 2.1).

TMT Transient Message Time (section 5.1).

TWE Time Warp Executive (section 2.2).

VRT Virtual Receive Time (section 2.3).

VST Virtual Send Time (section 2.3).

Bibliography

[1] Hallo Ahmed, Robert Ronngren, and Rassul Ayani. Impact of event scheduling on performance of time warp parallel simulations. In *Proceedings of the Twenty-seventh Annual Hawaii International Conference on System Sciences*, January 1994.

[2] Argonne National Laboratory. The Message Passing Interface (MPI) standard. http://www-unix.mcs.anl.gov/mpi.

[3] Duane Ball and Susan Hoyt. The adaptive Time-Warp concurrency control algorithm. In *Proceedings of the SCS Multi Conference on Distributed Simulation*, pages 174–177. The Society for Computer Simulation, 1990.

[4] Steven Bellenot. Performance of a riskfree Time Warp Operating System. In *Proceedings of the 7th Workshop on Parallel and Distributed Simulation*, pages 155–158. The Society for Computer Simulation, 1993.

[5] T. D. Blanchard, T. W. Lake, and S. J. Turner. Cooperative acceleration : robust conservative distributed discrete event simulation. In *Proceedings of the 8th Workshop on Parallel and Distributed Simulation*, pages 58–64, 1994.

[6] Randy Brown. Calendar queues : A fast O(1) priority queue implementation for the simulation event set problem. *Communications of the ACM*, 31(10), October 1988.

[7] R. Bryant. Simulation of packet communication architecture computer systems. Technical Report MIT/LCS/TR-188, MIT, November 1977.

[8] K. M. Chandy and J. Misra. Distributed simulation : A case study in design and verification of distributed simulation. *IEEE Transactions on Software Engineering*, 5(5):440–452, September 1979.

[9] Eunmi Choi and Dugki Min. Event scheduling schemes for time warp on distributed systems. In *Proceedings of the 1996 Winter Simulation Conference*, pages 661–668, 1996.

[10] John Cleary, Fabian Gomes, Brian Unger, Xiao Zhonge, and Raimar Thudt. Cost of state saving and rollback. In *Proceedings of the 8th Workshop on Parallel and Distributed Simulation*, pages 94–101. The Society for Computer Simulation, 1994.

[11] S. R. Das. Estimating the cost of throttled time warp. In *Proceedings of the 10th Workshop on Parallel and Distributed Simulation*, pages 186–189. The Society for Computer Simulation, 1996.

[12] Samir R. Das and Richard M. Fujimoto. A performance study of the cancelback protocol for Time Warp. In *Proceedings of the 7th Workshop on Parallel and Distributed Simulation*, pages 135–142. The Society for Computer Simulation, 1993.

[13] Samir R. Das and Richard M. Fujimoto. An adaptive memory management protocol for Time Warp parallel simulation. In *Proceedings of the 1994 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 201–210. ACM, 1994.

[14] Phillip M. Dickens, Phillip Heidelberger, and David M. Nicol. A distributed memory LAPSE : Parallel simulation of message-passing programs. In *Proceedings of the 8th Workshop on Parallel and Distributed Simulation*, pages 32–38. The Society for Computer Simulation, 1994.

[15] L. M. D'Souza, X. Fan, and P. A. Wilsey. pGVT : An algorithm for accurate GVT estimation. In *Proceedings of the 8th Workshop on Parallel and Distributed Simulation*, pages 102–109. The Society for Computer Simulation, 1994.

[16] A. Ferscha and G. Chiola. Self-adaptive logical processes: the probabilistic distributed simulation protocol. In *Proceedings of the 27th Annual Simulation Symposium*, pages 78–88. IEEE, 1994.

[17] Alois Ferscha. Probabilistic adaptive direct optimism control in Time Warp. In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*, pages 120–129. The Society for Computer Simulation, 1995.

[18] Alois Ferscha and Johannes Luthi. Estimating rollback overhead for optimism control in Time Warp. In *Proceedings of the 28th Annual Simulation Symposium*, pages 2–12. IEEE, 1995.

[19] Alois Ferscha and Michael Richter. Time warp simulation of timed petri nets : Sensitivity of adaptive methods. Technical report, Universitat Wien, 1997.

[20] R. M. Fujimoto. Time Warp on a shared memory multiprocessor. *Transactions of the Society for Computer Simulation*, 6(3):211–239, July 1990.

[21] Richard Fujimoto and Peter Hoare. HLA RTI performance in high speed LAN environments. In *Fall Simulation Interoperability Workshop*, September 1998.

[22] Richard M. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, October 1990.

[23] Richard M. Fujimoto. Performance of time warp under synthetic workloads. In *Proceedings of the SCS Multi Conference on Distributed Simulation*, pages 23–28, 1990.

[24] K. Ghosh, K. Panesar, R. M. Fujimoto, and K. Schwan. PORTS : A parallel, optimistic, real-time simulator. In *Proceedings of the 8th Workshop on Parallel and Distributed Simulation*, pages 24–31. The Society for Computer Simulation, 1994.

[25] Fabian Gomes, Brian Unger, and John Cleary. Language based state saving extensions for optimistic parallel simulation. In *Proceedings of the 1996 Winter Simulation Conference*, pages 794–800, 1996.

[26] Fabian A. F. B. Gomes. *Optimizing Incremental State Saving and Restoration.* PhD thesis, The University of Calgary, April 1996.

[27] Michial Gunter. Understanding supercritical speedup. In *Proceedings of the 8th Workshop on Parallel and Distributed Simulation*, pages 81–87. The Society for Computer Simulation, 1994.

[28] Donald O. Hamnes and Anand Tripathi. Investigations in adaptive distributed simulation. In *Proceedings of the 8th Workshop on Parallel and Distributed Simulation*, pages 20–23. The Society for Computer Simulation, 1994.

[29] David Jefferson. Distributed simulation and the Time Warp Operating System. *ACM Operating System Review*, pages 77–93, November 1987.

[30] David Jefferson. Virtual time II: Storage management in distributed simulation. In *Proceedings of the 9th Annual ACM Symposium on Distributed Computing*, pages 75–89. ACM, 1990.

[31] David Jefferson and Peter Reiher. Supercritical speedup. In *Proceedings of the 24th Annual Simulation Symposium*, pages 159–168. IEEE, 1991.

[32] David R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, pages 404–425, July 1985.

[33] Yi-Bing Lin and Edward D. Lazowska. Determining the global virtual time in a distributed simulation. In *1990 International Conference on Parallel Processing*, pages 201–209. IEEE, 1990.

[34] Yi-Bing Lin, Bruno R. Preiss, Wayne M. Loucks, and Edward D. Lazowska. Selecting the checkpoint interval in Time Warp simulation. In *Proceedings of the 7th Workshop on Parallel and Distributed Simulation*, pages 3–10. The Society for Computer Simulation, 1993.

[35] B. D. Lubachevsky. Several unsolved problems in large-scale discrete event simulation. In *Proceedings of the 7th Workshop on Parallel and Distributed Simulation*, pages 60–67. The Society for Computer Simulation, 1993.

[36] Boris Lubachevsky, Adam Schwartz, and Alan Weiss. Rollback sometimes works ... if filtered. In *Winter Simulation Conference*, pages 630–639. The Society for Computer Simulation, 1989.

[37] Boris Lubachevsky and Alan Weiss. An analysis of rollback-based simulation. *ACM Transactions on Modelling and Computer Simulation*, pages 154–193, April 1991.

[38] Boris D. Lubachevsky. Scalability of the bounded lag distributed discrete event simulation. In *Proceedings of the SCS Multi Conference on Distributed Simulation*, pages 100–107. The Society for Computer Simulation, 1989.

[39] Dale E. Martin, Philip A. Wilsey, and Timothy J. McBrayer. *WARPED (version 0.5)*. University of Cincinnati, 1995.

[40] Edward Mascarenhas, Felipe Knop, and Vernon Rego. Minimum cost adaptive synchronization: Experiments with the PARASOL system. In *Proceedings of the 1997 Winter Simulation Conference*, pages 452–459, 1997.

[41] Friedemann Mattern. Efficient algorithms for distributed snapshots and global virtual time approximation. Technical report.

[42] Avinash C. Palaniswamy. *Dynamic Parameter Adjustment to Speedup Time Warp Simulation*. PhD thesis, The University of Cincinnati, 1994.

[43] Avinash C. Palaniswamy and Philip A. Wilsey. An analytical comparison of periodic checkpointing and incremental state saving. In *Proceedings of the 7th Workshop on Parallel and Distributed Simulation*, pages 127–134. The Society for Computer Simulation, 1993.

[44] Avinash C. Palaniswamy and Phillip A. Wilsey. Parameterized time warp (PTW): An integrated adaptive solution to optimistic PDES. *Journal of Parallel and Distributed Computing*, 37(2), September 1996.

[45] Kiran S. Panesar and Richard M. Fujimoto. Adaptive flow control in time warp. In *Proceedings of the 11th Workshop on Parallel and Distributed Simulation*, pages 108–115. The Society for Computer Simulation, 1997.

[46] David Jefferson Peter L. Reiher, Frederick Wieland. Limitation of optimism in the time warp operation system. In *Proceedings of the 1989 Winter Simulation Conference*, pages 765–770, 1989.

[47] P. L. Reiher and D. Jefferson. Virtual time based dynamic load management in the Time Warp Operating System. In *Proceedings of the SCS Western Multi Conference on Distributed Simulation*, pages 103–111. The Society for Computer Simulation, 1990.

[48] Paul F. Reynolds Jr. A shared resource algorithm for distributed simulation. In *9'th International Symposium on Computer Architecture*, pages 259–266. IEEE, 1982.

[49] Paul F. Reynolds Jr. A spectrum of options for parallel simulation. In *Winter Simulation Conference*, pages 325–322. The Society for Computer Simulation, 1988.

[50] Robert Ronngren, Michael Liljenstam, Rassul Ayani, and Johan Montagnat. Transparent incremental state saving in Time Warp parallel discrete event simulation. In *Proceedings of the 10th Workshop on Parallel and Distributed Simulation*, pages 70–77. The Society for Computer Simulation, 1996.

[51] Robert Sedgewick. *Algorithms in C.* Addison Wesley, 1990.

[52] Andrew F. Seila. Introduction to simulation. In *Proceedings of the 1995 Winter Simulation Conference*, pages 7–15. The Society for Computer Simulation, 1995.

[53] D. Sleator and R. Tarjan. Self-adjusting binary search trees. *Journal of the ACM*, 32(3):652–686, 1985.

[54] Lisa M. Sokol and Brian K. Stucky. MTW : Experimental results for a constrained optimistic scheduling paradigm. In *Proceedings of the SCS Multi Conference on Distributed Simulation*, pages 169–173. The Society for Computer Simulation, 1990.

[55] Hussam M. Soliman and Adel S. Elmaghraby. The aggressive adaptive-risk approach for parallel simulation. *Transactions of the Society for Computer Simulation*, 13(3):117–124, 1996.

[56] Tapas K. Som and Robert G. Sargent. A probabilistic event scheduling policy for optimistic parallel discrete event simulation. In *Proceedings of the 12th Workshop on Parallel and Distributed Simulation*, pages 56–63, 1998.

[57] Sudhir Srinivasan, Margaret J. Lyell, Paul F. Reynolds Jr., and Jeff W. Wehrwein. Implementation of reductions in support of PDES. In *Proceedings of the 12th Workshop on Parallel and Distributed Simulation*, 1998.

[58] Sudhir Srinivasan and Paul F. Reynolds Jr. Non-interfering GVT computation via asynchronous global reductions. In *Proceedings of the 1993 Winter Simulation Conference*, 1993.

[59] Sudhir Srinivasan and Paul F. Reynolds Jr. Adaptive algorithms vs. Time Warp : An analytical comparison. In *Proceedings of the 1995 Winter Simulation Conference*, pages 666–673. The Society for Computer Simulation, 1995.

[60] Sudhir Srinivasan and Paul F. Reynolds Jr. NPSI adaptive synchronization algorithms for PDES. In *Proceedings of the 1995 Winter Simulation Conference*, pages 658–665. The Society for Computer Simulation, 1995.

[61] Sudhir Srinivasan and Paul F. Reynolds Jr. Super-criticality revisited. In *Proceedings of the 9th Workshop on Parallel and Distributed Simulation*, pages 130–136. The Society for Computer Simulation, 1995.

[62] J. S. Steinman. Discrete-event simulation and the event horizon. Technical report, Jet Propulsion Laboratory, California Institute of Technology, 4800 Oak Grove Drive, Mail Stop 525-3960, Pasadena, CA 91109, 1993.

[63] J. S. Steinman. Incremental state saving in SPEEDES using C++. Technical report, Jet Propulsion Laboratory, California Institute of Technology, 4800 Oak Grove Drive, Mail Stop 525-3960, Pasadena, CA 91109, 1993.

[64] J. S. Steinman. SPEEDES : A unified approach to parallel simulation. Technical report, Jet Propulsion Laboratory, California Institute of Technology, 4800 Oak Grove Drive, Mail Stop 525-3960, Pasadena, CA 91109, 1993.

[65] Jeff S. Steinman. Breathing Time Warp. In *Proceedings of the 7th Workshop on Parallel and Distributed Simulation*, pages 109–118. The Society for Computer Simulation, 1993.

[66] Jeff S. Steinman. Discrete-event simulation and the event horizon. In *Proceedings of the 8th Workshop on Parallel and Distributed Simulation*, pages 39–49. The Society for Computer Simulation, 1994.

[67] Jeff S. Steinman and Frederick Wieland. Parallel proximity detection and the distribution list algorithm. In *Proceedings of the 8th Workshop on Parallel and Distributed Simulation*, pages 3–11. The Society for Computer Simulation, 1994.

[68] Seng Chaun TAY, Yong Meng TEO, and Siew Theng KONG. Speculative parallel simulation with an adaptive throttle scheme. In *Proceedings of the 11th Workshop on Parallel and Distributed Simulation*, pages 116–123. The Society for Computer Simulation, 1997.

[69] Stephen J. Turner and Ming Q. Xu. Performance evaluation of the Bounded Time Warp algorithm. In *Proceedings of the SCS Multi Conference on Parallel and Distributed Simulation*, pages 117–126. The Society for Computer Simulation, Jan 1992.

[70] Darrin West. Lazy rollback and lazy reevaluation. Master's thesis, The University of Calgary, 1988.

[71] Darrin West and Kiran Panesar. Automatic incremental state saving. In *Proceedings of the 10th Workshop on Parallel and Distributed Simulation*, pages 78–85. The Society for Computer Simulation, 1996.

[72] Frederick Wieland. The threshold of event simultaneity. In *Proceedings of the 11th Workshop on Parallel and Distributed Simulation*, pages 56–59. The Society for Computer Simulation, 1997.

[73] Kenneth R. Wood and Stephen J. Turner. A generalized carrier-null method for conservative parallel simulation. In *Proceedings of the 8th Workshop on Parallel and Distributed Simulation*, pages 50–57. The Society for Computer Simulation, 1994.

[74] Z. Xiao and B. Unger. Report on warpkit - performance study and improvement. Technical Report Technical Report 98-628-19, Computer Science Department, University of Calgary, May 1995.

[75] Z. Xiao, B. Unger, R. Simmonds, and J. Cleary. Scheduling critical channels in conservative parallel discrete event simulation. In *Proceedings of the 13th Workshop on Parallel and Distributed Simulation*, May 1999.